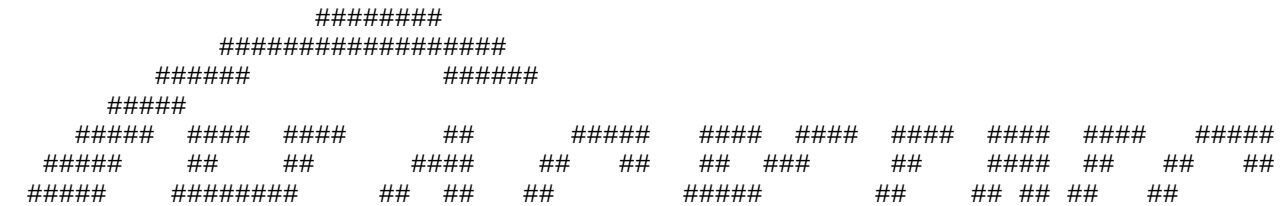


Volume 1, Issue #4
October 5, 1992

Editor's Notes:
by Craig Taylor (duck@pembvax1.pembroke.edu)

My apologies about this issue being posted later than was mentioned in a preview post on comp.sys.cbm newsgroup. Due to some problems with coding, school and Murphy's law the issue had to be delayed until now.

I have asked the system admin's at my site concerning a mail-server but they said they did not have enough man-power (go figure) to get somebody to run it. I will be implementing a mail-server system in my account in the near future for retrieval of programs and back-issues. I'll post descriptions of how to use it it the next issue of C= Hacking as well as on the newsgroup comp.sys.cbm when I finish writing it.

In this issue of C= Hacking we also start on an ambitious task: Developing a game for both the C128 and C64 modes that includes all of the features found in commercial games. Take a look in the Learning ML Column for more information.

Also, The article concerning the 1351 mouse has _again_ been delayed due to time constraints. Rest assured that it will be in the next issue of C= Hacking.

If you are interested in helping write for C= Hacking please feel free to mail duck@pembvax1.pembroke.edu (or duck@handy.pembroke.edu). We're always looking for new authors on almost any subject, software or hardware.

==========

Also note that this issue and prior ones are available via anonymous ftp from ccosun.caltech.edu under pub/rknop/HACKING.MAG.

==========

NOTICE: Permission is granted to re-distribute this "net-magazine", in whole, freely for non-profit use. However, please contact individual authors for permission to publish or re-distribute articles seperately.

*** AUTHORS LISTED BELOW RETAIN ALL RIGHTS TO THEIR ARTICLES ***

==========

In this issue:

Learning ML - Part 4

In the next issue we'll embark on a project of making a space invaders style game for the C=64/128 from scratch using custom characters, interrupt-driven music, animation, using the joystick, mouse or keyboard. The C64 and C128 versions will be developed con-currently, each program taking advantage of

the machine's capabilities. This is the first in a series - written by Craig Taylor.

The Demo Corner: FLI - more color to the screen

All of us have heard complaints about the color constraints on C64. FLI picture can have all of the 16 colors in one character position. What then is this FLI and how it is done ? Written by Pasi 'Albert' Ojala.

RS-232 Converter

This article details plan for a User port TO RS232 connector using just ONE IC and 4 capacitors. The circuit is included, and suggestions on alternative chips and parts are examined. Written by Warren Tustin

Introduction to the VIC-II

This article examines the VIC-II chip in detail and provides an explanation of the various registers associated with the chip. Written by Pasi 'Albert' Ojala.

LITTLE RED READER: MS-DOS file reader for the 128 and 1571/81 drives.

This article presents a program that reads MS-DOS files and the root directory of MS-DOS disks. This program copies files from disk to disk so two disk drives are required to use it (or a "virtual" drive). This scheme imposes no limit on the maximum size of a file to be transferred. The user-interface code is written in BASIC and presents a full-screen file selection menu. The grunt-work code is written in assembly language and operates at maximum velosity. Complete, explained code lisitings are included. By Craig Bruce.

==========

Learning Machine Language - Part 4

by Craig Taylor (duck@pembvax1.pembroke.edu)

```
+---------------------------+
| Space Invasion - Part 1   |
|                           |
| Programming: Craig Taylor |
| Graphics   : Pasi Ojala   |
| Music/Sound:              |
|                           |
+---------------------------+
```

I. Introduction

In this and future Learning Machine Language's we will develop a game called Space Invasion. The game will be similair to Space Indvaders and will run on the Commodore 64 or the Commodore 128 in 80 columns. It will feature all the "features" and "parts" that are found in commercial games with interrupt-driven music, custom character definitions, 100% machine language, multi-level game play, and input from the keyboard, joystick or mouse.

Note | I am looking for someone to help aid music composition that will -----+ be introduced in a later issue. Programming of the 6502 is helpful but not a requirement. Please email me at duck@pembvax1.pembroke.edu if you are interested.

Many thanks to Pasi Ojala for his work with the graphics in this program.

Also please note: This entire program has been assembled sucessfully with the Buddy-128 assembler for both the C=128 and C=64 version. Due to the length of the source files (over 1,500 lines) I'm not sure if Buddy-64 will handle it. Thus if you get errors during assembly, all I can say is: sorry.

If this is the case then the next issue will handle dividing the program and data up into segments which can then each be loaded seperatly.

II. Machine Notes

The Commodore 64 and 128 programs for Space Invasion will differ slightly, mostly in the following areas:

- custom character definition
- memory initialization / setup
- sound / music

Because the actual game play and the changes nesscesary between the areas listed above, we will use the Buddy Assembler notation for conditional assembly to allow the development of only one file containing the source code. In addition to conditional assembly most of the routines will be written as one with jumps to subroutines containing the C64 or 128 direct code as the algorithims are usually the same for each.

In addation there will be several source files and some miscellaneous include files for graphics and sound. For those of you who are or will be converting the assembly source over to a different assembler the conditional assembly directives .if (condition) will only be true if the condition is non-zero. Ie: if the symbol computer is defined as 128 then the following example illustrates it:

```
computer = 128
.if computer-64         ; non-zero answer so therefore
; 128 code goes here
.else
; 64 code goes here
.ife                    ; end the .if condition.
```

Also note that for much of the program we will _not_ be using the computer routines and instead be developing our own.

In addition the program will show you how to use IRQ interrupts to simplify programming. We will be using them to play music in the background on three voices (sound effects will temporarily pre-empt the third voice from playing). Also animation of characters will be done via the IRQ. A little background on interrupts for those of you who are a bit hazy on what they are or have never seen them before (Also try taking a look at Rasters: What they are and how to use them in C= Hacking #3 - While this does not necessarily cover what we are going to be using interrupts for it does describe them quite well.) Basically the computer generates an interrupt every 1/60th a second from a timer on the computer (usually from the CIA chip or the screen for those of you who are curious). The computer will save all the registers, jump to a subroutine - perform the instructions there (usually updating time, scanning the keyboard etc...) and then recall all the registers and return to the user program. This is an interrupt. An IRQ interrupt describes an interrupt that we can allow to be "turned on" and "turned off" - ie: we can temporarily disable it if we have to. A NMI interrupt describes an interrupt which we can _not_ temporarily disable -- we will not be using NMI interrupts in this program.

III. The Process

Part of what this series of articles is focused at is the development of being able to analyze programming tasks and break them down into smaller workable problems. Once these problems or subroutines are completed your original problem is solved.

Let's take this approach to Space Invasion:

Problem Statement: Build a Space Invader program called Space Invasion.

Usually, given a problem you have to re-work the problem statement to encompass all of what you want. Let's try again:

Problem Statement: Develop a Space Invader program called Space Invasion
---------- utilizing the 64 or 128 screen with interrupt driven
music / sound, and allowing input from the keyboard,
joystick or mouse.

Hmmm... The problem statement listed above is better but it has no real order; we have no clear idea of where to start and what we need to do. It does however tell us that we have the following sections:

- 64 / 128 Screen Handling
- Music / Sound
- Input Handling
- Game Driver (implied)

Let's think a bit more about each of these sections and what each will involve:

128 / 64 Screen Handling:

- Putting characters on screen.
- Initializing the Screen / Registers.
- Setting up the Custom Characters.
- Handling any Animation.

Music / Sound:

- Setting up the Sound Chip Registers.
- Playing a note read from Memory.
- Executing a Sound Effect.

Input Handling:

- Device Selection (keyboard, mouse, joystick).
- Keyboard Scanning.
- Mouse Scanning.
- Joystick Scanning.

Game Driver:

- Title Screen.
- Initialization of Memory.
- Level Setup.
- Movement of Aliens.
- Movement of Missles.
- Movement of Player.
- Collision Checking.
- Collision Handling.
- End of Level.
- Score Updating.
- End - Game handling.
- High Score Update.

Shrew! Long list 'eh? - Now you may have thought of some not listed above, and we may have possibly overlooked some crucial routines -- that's fine -- the above is just intended as a building block - a place to start coding from.

If we think of these as subroutines we can build a skeleton outline of the program - yet we need some order in how we call them. Obviously we aren't going to move the player until we scan the input and that requires prior device selection etc...

Hmm... Taking order into account we can re-state the problem as:

Problem Statement: Develop a game similair to Space Invaders called Space

---------------- Invasion by initializing memory, the display device, setting up Custom Characters, setting up the Music Registers and displaying the title screen. From there, select the input device and after that setup the current level. Next, while playing music in the background and scanning the input device, move the aliens, missles and player checking for collisions and taking appropriate action as required (player dies, score increases etc or what-not). After each level display if the player is dead, or set-up for the next level and repeat. When the game has ended update the high score if necessary.

Try saying that five times real fast! :-) But that problem statement is a whole lot better than the one we had at the beginning which simply said to develop a game.

IV. Not All At One Time - What We're Doing This Time

Now this program is too complex, (as seen by the problem statement above) to have in one article so this issue we'll concentrate on the basic main loop and the initialization of the Custom Characters and the title screen.

Originally, I was planning on updating and listing the revised code in each issue. However, due to space limitations and the enormity of the program currently (1,500+ lines!!) it will be placed for anonymous ftp at ccosun.caltech.edu under the directory: pub/rknop/HACKING.MAG.

V. The Main Loop

What is a main loop? Basically it's where everything gets done. It calls other subroutines and keeps repeating until certain criteria are met - usually when the player requests to exit the game. However, inside you'll find inner loops for level play etc.

Our main loop for this program will be:

```
--------------------------------------------------------------------------------
;; * Main Loop - This should be the last section in the source code.
;
; Main Loop
;

main'loop = *
               jsr memory'setup        ; Set-Up memory.
               jsr display'setup       ;  "  "  display.
               jsr char'setup          ;  "  "  custom character display.
               jsr music'setup         ;  "  "  music chip.
               jsr title'screen        ; Display the title screen.
               jsr select'input        ; Select Input Device.

level'loop = *
               jsr play'music          ; Start the music playing.
               jsr setup'level         ; Setup the current level.

             - jsr alien'move          ; Move aliens
               jsr missle'move         ;  "   missles
               jsr player'move         ;  "   player
               jsr check'collision     ; Check for collisions
               ldx collision'flag      ; Check collision flag.
               beq -
```

```
                dex                     ; Decrease .X by 1 so if X was 1 then
                beq player'die          ;    it's now 0 so we know player died.
                dex                     ; Decrease .X again so if X was 2 then
                beq alien'die'sound     ;    it's now 0 so we make alien death.
                jsr end'level           ; If we got here - than end of level.
                jsr wait'next           ; Wait for next keypress.
                jsr increase'level      ; increase level #.
                sec                     ; And go back....
                bcs level'loop

alien'die'sound = *
                jsr make'alien'sound    ; make alien sound.
                sec                     ; set carry
                bcs -                   ; and jump back.

player'die      jsr show'player'die     ; Show it on-screen.
                lda lives               ; Check # of lives.
                beq end'of'game         ; If 0 the end-of-game.
                bne level'loop          ; go back and re-start level.
                brk                     ; If we get here - than an error.

end'of'game     jsr end'game'screen     ; Show end-of-game screen.
                jsr high'score'update   ; Update the high score if need-be.
                jsr wait'next           ; Wait for next-game selection.
                lda quit
                beq +
                jsr setup'level'1       ; Set-Up first level.
                sec
                bcs level'loop          ; and start playing it.

              + jmp quit'game
;
; End of Main Loop
;
--------------------------------------------------------------------------------
```

Some of the routines listed above we will later replace with actual code. It's much easier to see:

```
                inc level
```

than to see a

```
                jsr increase'level
```

and try to hunt down the code. I've included them in for now so that we can have a better idea of what is going on.

In the file: invasion.src most of the statements above are commented out. Once we write the routines we'll un-comment them. For now, this serves to still remind us of the routines we need to write.

Also there are a couple of programming tricks that I used in the main loop that probably need some clarifying.

When handling the collisions the .X register is loaded with the result of the collision checking - $00 = no collisions, $01 = player died, $02 = alien died, $03 = end of level. Anytime a load to a register is done the flags are automatically set as if you had compared it to 0 - hence we can ldx the collision flag and immediately branch if equal to zero for no collisions. In addition to the load anytime the .X or .Y registers are incremented or decremented an implicit comparison to zero is performed. So if the .X register is 1 previously, we decrement it then it will be zero and our BEQ instruction will branch. If it's two then it will be one and we can continue like this.

[NOTE: Technically it's not a real comparison to zero but calling it a

comparison to zero servers our purpose here. The only significant difference would be in the effect of the carry flag which is insignificant in our code segment here.]

Also in several locations are the two instructions:

```
sec
bcs [label]
```

What these are doing are simply programming style - they could be substituted with JMP [label] - however they offer advantages over JMP. They take up the a larger amount of execution time, however they are relocatable so any mucking around / moving sections of code during debugging will be less likely to crash. Using other flags are also valid -- the use of which flag (I prefer the carry flag) is usually dependent on the programmer. Geos defines a similair macro called BRA (branch always) which is equivlent to:

```
clv
bvc [label]
```

Note that the above is just programming style, held over from my programming in assembly days. The use of JMP is probably preferable in terms of execution and also in being able to branch more than 127 bytes away (the branch instructions only have a range of +128/-127).

VI. Custom Characters

Since we're writing for each of the seperate modes (64 mode, 128 mode) we have to take a look at the differences between the VIC chip (64 mode) and the 8563 chip in the 128.

The Vic-Chip

The character sets in the VIC chip are defined as in the example below of the character code $00 "@" (all references are to screen "poke" codes - not print codes).

```
       .byt #%00111100    Try holding the page (or moving away from the
       .byt #%01100110    screen) and taking a look at the patterns the 1's
       .byt #%01101110    and 0's make. Each character is thus defined as
       .byt #%01101110    eight bytes who's bit patterns define it.  Having a
       .byt #%01100000    total of 256 characters available makes it
       .byt #%01100010    neccesary to set aside a total of 2,048 bytes.
       .byt #%00111100
       .byt #%00000000
```

Now, instead of designing all 256 character sets we'll just take advantage of the fact that the letters and numbers we want will already be there -- we'll just copy them from the ROM set into RAM, modify some of the other characters to reflect what we want and then tell the VIC chip to look at RAM to get the character set definitions.

There are some problems with copying the 'system' characters, however. The Commodore 64 usually masks out the character set and typically it is only available to the VIC chip so that more space can be present for user programs and such. It also takes up the section of memory that the I/O block in $d000-$dfff does so that switching it in while interrupts are enabled is sure to result in a crash.

We're also going to be doing a few things that you may not expect -- instead of copying all 256 characters - we're gonna _just_ copy the first 128. This will give us all of the normal characters as the last 128 are the reverse-video counterparts to the first 128 characters. We're doing this to conserve

space and because we really don't need that many characters defined.

Also location $01 contains what $d000-$dfff holds and we will have to modify bit 2 to switch the character ROM in. Hence, the following program code is used to copy the character set:

```
-------------------------------------------------------------------------------
copy'chars = *                  ; must be run w/ interrupts disabled
               lda $01          ; register 1 = the control to switch in the char.
                                ; rom.
               pha              ; save it as we'll later need to sta' it back.
               and #%11111011   ; Bit 2 controls it - clear it to switch it in.
               sta $01          ; and make it so we can read it in.
               lda #>$3000      ; move chars to $3000
               sta dest+1
               lda #>$d800      ; from $d800 (start of char set) (lower-case)
               sta src+1
               ldy #$00         ; lo-bytes of both src, dest = $00.
               sty src
               sty dest
               ldx #$10         ; copy 2k of data.
             - lda (src),y      ; copy byte.
               sta (dest),y
               iny
               bne -            ; continue until .Y = 0.
               inc src+1        ; increase source & dest by 256
               inc dest+1
               dex              ; decrease .X count.
               bne -            ; if non-zero then continue copying, else
               pla              ; restore value of $01
               sta $01          ; and put back.
               lda $d018        ; set VIC-chip address.
               and #$f1         ; to show char set.
               ora #$0c
               sta $d018        ; and finally tell VIC where the char set is...
               rts              ; and return.
-------------------------------------------------------------------------------
```

Note that we still need to change the actual characters we're gonna be using. That will be handled in the section after next: Changing the Characters as there is a great deal of similarity between the 128 and 64 implementations.

The 8563 Chip

The 8563 80-Column chip usually has 16k or 64k Ram attatched to the chip which the CPU does not have direct control over. It has to direct the 8563 to store and retrieve values to that memory. What makes control over that memory all the more difficult is the fact that the 8563 only has two lines or addresses that the CPU can control.

The 8563 has a character set in much the same way the VIC chip does, save one exception - each character set can have up to 16 lines. Normally, the last eight lines are filled with $00 and are not shown. (Provisions can be made to have 8x16 characters but it is not needed for this game and thus, will not be shown - For more information See C= Hacking Issue #2: 8563: An In-Depth Look.) Thus the algorithim is similair to the C=64 but 8 zero-bytes will need to be written at the end of every eight bytes read.

However, the 8563 does make things easier for us! - When the computer is first turned on a copy of the Character Set from ROM is copied into the 8563. The 8563 has no ROM Character Set associated with it and thus we are able to just simply modify the character set that is in the 8563 memory instead of copying it over. Because of this no routine will be presented to copy a character set into the 8563 memory, rather the discussion of copying individually defined characters will take place in the next section. The C=128 also makes

life even easier for us at the end when we will exit the program, modifying the character set back to the "standard" Commodore character set by a routine in the KERNAL that will copy the characters back. We'll take a look at it closer when we write the exit routine.

Also note that since the 8563 chip supports the 80 column screen we will be defining two characters that can be placed side by side for each alien so that the playing field will be similair to the C64 version. However, for the title screen we will be switching the 8563 into a "40 column" mode to make programming easier, in addition to expanding the character bit-mapped logo.

Changing the Characters

A lot of the times you'll find yourself re-using subroutines and code that you have previously created, gradually, over a period of time building up a library of routines. When thinking through the purpose and intent of this routine I thought about possibly building it so it would read a table and change the character set based on that table. The 64/128 character sets would be the same - this routine would automatically generate the eight additional bytes needed by the 8563 if need-be and it would call the appropriate storage routine - store to either the 8563 or the computer memory.

Now you may be asking why would you want to store to the computer memory in 128 mode? Why not just have two seperate versions? - Yes - that could be possible but I'm implementing it this way because in the future I may see a need to define custom characters in 128 mode for the 40 column screen. This way I can just extract the routine, pop it into my program and I've got that section of the code complete.

This is what I was thinking of for the data table:

```
    .byt 1 = 8563, 0 = comp. memory.
    .word address ; address base of char-set in computer or 8563 memory.
    .byte char #  ; (to start)
    .byte # of chars to define
    .byte # of characters to define
    .byte data,data,....,data8 ; character data.
    .byte data,data,....,data8 ; character data. etc....
    . . .
```

Entrance into the routine will consist of .AY holding the location of the table. We will keep the address of the table and keep incrementing it as we go along in z-page locations.

```
-------------------------------------------------------------------------------
      install'char = *
              sta zp1                 ; save .ay in table address
              sty zp1+1
              ldy #$00                ; read computer mode.
              jsr get'byte
              sta mode
              jsr get'byte            ; get address base.
              sta adr
              jsr get'byte
              sta adr+1
              jsr get'byte            ; get number of characters to copy.
              sta numb
              jsr get'byte            ; get next character #.
              sta wrk                 ; save in temp. location.
              lda #$00
              sta wrk+1
              asl wrk                 ; shift left x3 times = *8
              rol wrk+1
```

```
              asl wrk
              rol wrk+1
              asl wrk
              rol wrk+1
              lda mode              ; if for 8563 then multiply 1 more time.
              beq +
              asl wrk
              rol wrk+1
            + lda adr               ; add character address in.
              clc
              adc wrk
              sta wrk
              lda adr+1
              adc wrk+1
              sta wrk+1             ; address now calculated
              jsr setadrs           ; set address in proper chip
  loop'install ldx #$08             ; copy 8 bytes.
            - jsr get'byte
              jsr writebyte         ; write out byte.
              dex
              bne -
              lda mode              ; if 128 then fill out 8 more $00 bytes.
              beq +
              lda #$00
              ldx #$08
            - jsr writebyte
              dex
              bne -
            + dec numb
              bne loop'install
              rts
```

What? We have thre subroutines : writebyte, setadrs, and get'byte that we haven't examined yet. These are going to be the routines that are dependant on the computer type. Also, writebyte will require that .XY not be disturbed; setadrs requires that .Y not be disturbed hence the following:

```
       setadrs tya              ; save .yx
               pha
               txa
               pha
               lda mode         ; check computer type.
               beq +            ; if C=64, then jump ahead.
               ldx #18          ; VDC register - current memory address hi
               lda wrk+1        ; get address hi
               jsr wr'vdc
               ldx #19          ; VDC register - current memory address lo
               lda wrk          ; get address lo
               jsr wr'vdc
             + pla              ; restore .XY
               tax
               pla
               tay
               rts              ; and return.
```

Note that we really don't need a setadrs for the C=64 -- we can just index off (wrk) in the writebyte routine which follows:

```
  writebyte    sta temp         ; save as we need it later.
               txa              ; Save .XY
               pha
```

```
              tya
              pha
              lda mode        ; now check computer type.
              beq +           ; if c64 jump ahead
              lda temp        ; recall temp.
              jsr wr'vram
              sec
              bcs ++          ; jump ahead
            + ldy #$00        ; C64 / y-index = $00
              lda temp        ; get value
              sta (wrk),y     ; store
              inc wrk         ; now increase address
              bne +
              inc wrk+1
            + pla             ; now return after recalling .XY
              tay
              pla
              tax
              rts             ; and return.
```

Note that the following routine is fairly short but it is called numerous times within the routines that use data tables such as install'char, write'txt and write'col.

```
  get'byte = *
              lda (zp1),y
              iny
              bne +           ; if zero then increase zp1 hi
              inc zp1+1
            + rts
```

Not bad 'eh? A quick note: The instructions: PLA, TAY, PLA, TAX, PHA, etc.. are routines that Push or Pull (pha,pla) the .A onto the stack. The TAY, TAX, TXA, TYA are instructions that transfer a register to another (ie: the TAY transfers the A register to .Y, TXA transfers .X to .A etc...) By using the combination of these with the stack we can save the registers and later re-call them so that they are the same when we entered the routine. The stack is usually a "mystery" item to new programmers of the 6502 series. Basically it's just like any other stacks in the real world - the last item thrown (I'm non-practicing perfectionist so I throw stuff.. ;-)) or pushed on the stack will the first item removed or pulled from the stack. For example I've got a stack of books sitting near me :

```
              Mapping the Commodore 128
              128 Internals
```

and I'm holding Mapping the Commodore 64 in my hands. If I push (or toss) the book onto the stack (and hopefully hit the stack instead of the floor) I'll have the following stack:

```
              Mapping the Commodore 64
              Mapping the Commodore 128
              128 Internals
```

and it should be easy to see that if I "pull" the next book off the stack that I'll get the Mapping the Commodore 64 book. The next book to be "pull"ed after that would be the Mapping the Commodore 128 book. This idea can be applied to the 6502 stack -- It will keep storing values (up to 256) when you "push" them on (via the PHA instruction) and will retrieve the last value stored when you "pull" them off (via the PLA instruction). Another PLA instruction would return the next value that had been stored.

The Character Bitmaps

Pasi Ojala is to be credited with all the graphics and many thanks go out to him.

The game logo is made up of 120 custom defined characters that will be printed in the following manner (on the 128 screen they will be centered).

(in reverse video)...

```
ABCDEFGH . . . [up to 40 characters]
IJKLMNOP
QRSTUVWX
```

and everything will line up.

So that it will look like a "mini-bitmap". We could have used bitmap mode and made a very nice looking title screen but that would have involved switching and allocating memory for the bitmap, etc . . . On both the 8563 and the VIC that involves a bit more work and so custom characters will be used for the title screen. The regular letter and numeric characters will be available so that we can display credits and game instructions below the logo.

Now - in the program listing we could list them as binary #'s and that would make editing them very easy but we're gonna use their decimal representation in the program listing.

The characters are defined similair to the logo except they are treated as single characters. In the 128 version due to the 80 column screen we are going to use two characters side by side to simulate one alien so that the playing field will be similair to the C64 version. In addation, during the main loop we will modify the character sets to support animation of the aliens. In the data listing there is a reference to "frames" - for each of the aliens there are 8 differant frames.

Oh! - There will be more characters defined in the future. Right now I'm mainly interested in getting some base characters down so you can see how custom characters are implemented. When we start setting up different levels and such we'll add more characters then. Currently the custom characters are not used - only the characters for the logo. For those of you who are curious try installing the characters via install'char and taking a look at the aliens.

VII. Title Screen

The title screen is usually a lead-in to the actual game and it's aim is to tell the player how to play the game, any available options and p'haps present a nice graphic or two to "wow" the user into playing. In addation, the main musical theme can be introduced here to unify the game-playing. The discussion below does not take into account color but rest-assured we will be using varying colors in the title screen. The format for the color data will be almost identical to the title screen format except it will be structured via the following:

```
.word address
.byte num_of_chars to put color ($00= end of data)
.byte color_value
```

The routine (color'text) can be found in the source listings at the end of this article. Because of the similarity between it and write'text it is not discussed in this article.

Title Screen BackGround

The title screen I envisoned as a bordered screen (using the normal C= character set - ie: C= A,S,Z,X on the keyboard) with our bitmap in the middle and under-neath it a short description of the game and game-play instructions.

Now this is my idea of the screen layout (rough drawing as we're not using the actual screen dimensions):

```
   +------------------------------------------------------------+
   | -LOGO ------------------------------------------------------ |
   | --------------x 3 lines------------------------------------ |
   | ------------------------------------------------------------ |
   |                                                            |
   |                  Space Invasion C64/128                    |
   |                    Programming : Craig Taylor              |
   |                    Graphics    : Pasi Ojala                |
   |                    Sound       : ???????????               |
   |                                                            |
   | ------------------------------------------------------------ |
   | To Play:                                                   |
   |    Use joystick in port 2, mouse in port 1 or keyboard:    |
   |            A - Left, Z - Right   Space - Fire              |
   |              F1 - Restart                                  |
   |                                                            |
   +------------------------------------------------------------+
```

Title Screen Formatting

We come into a problem here -- the screen is some 1000 characters on the C64, and 2000 characters for the C=128. It would be extremely wasteful to store that many characters in memory just to reproduce a title screen - and most of them consisting of spaces at that!!

What we'll do is to just specify the address on screen, the # of characters and then list the characters. It will be similair to our custom character table driver above but will be different enough that a new routine is warrented. We will however use the two subroutines writebyte and setadrs that were developed in the previous routine. The data will look like the following:

```
     .word address
     .byte num_of_chars ($00= end of data)
     .ascii "text"
     .byte address .... etc....
```

and we'll enter with .AY containing the address of the table.

So basically we come up with the following:

```
-------------------------------------------------------------------------------
      write'txt = *
              sta zp1         ; save .ay in table address
              sty zp1+1
              ldy #$00
  loop'w'text = *
              jsr get'byte   ; set address.
              sta wrk
              jsr get'byte
              sta wrk+1
              jsr get'byte    ; get # of chars to write out.
              cmp #$00
              beq +           ; if zero then exit.
              tax
```

```
              jsr setadrs     ; set address to wrk,wrk+1
            - jsr get'byte
              jsr writebyte   ; write out byte.
              dex
              bne -
              sec
              bcs loop'w'text ; this is an absolute jump to loop
            + rts             ; return.
```

This is similair to our previous routine, and was in fact copied and modified from the previous routine.

VIII. Debugging

Now, not all programs are perfect, and during the development of this portion of the game there were several errors found. Tracing an error in Machine/Assembly-Language is like trying to find a grammatical error in a language you don't know. ;-) But seriously, there are several ways to track down errors in your code.

1 - Try tracing it through by hand playing "What if I were the computer" and following what each register does.

2 - Are you switching the LoHi order of variables? Ie: is it lda #< or lda #>??

3 - Set BRK points and run the program / subroutine within a machine language monitor and make sure the registers / memory locations contain the values that they should. If not, find out why.

4 - Try to simplify your code in terms of programming ease - Make the assembler do the work for you - it's a lot less likely to make errors than you are.

5 - Think logically!!!

6 - Change something at random and pray.

I can't stress numbers 3 and 5 enough. During the writing of the install'char routine there were numerous bugs that were eventually tracked down by setting a BRK instruction further along in the code and seeing exactly what the register / memory locations were. Also the use of temporary load and store instructions into "safe" regions of memory helped me monitor what some of the values were.

For example, at one point I had a section of code similair to the following:

```
            clc
            lda value
            adc data
            bne +
            inc data+1
          + [.... ]
```

And it's purpose was to add value to data. Now I've found simple errors are usually found last, after complex errors. And not until a set a break point like:

```
            clc
            lda value
            adc data    <-----Missing Instruction after here-------------+
```

```
            bne +                                                    |
            inc data+1                                               |
          + BRK                                                      |
            [.... ]                                                  |
                                                                     |
  did I actually figure out that I was missing the STA DATA instruction --+
```

So, when writing, modifying, and trying to debug code try to take your time and isolate every possible problem. Also don't be afraid to stop the code mid-stream as in the above with use of the BRK. You can always remove it (and probahly should) in the final code and it serves as a very valuable debugging tool with the aid of a machine-language monitor.

IX. Memory Map Considerations

Before you start a program it's a good idea to consider where in memory you will have everything. Now we've already started some of the program above and just blindly picked numbers at random it seemed like $3000 for the character set for the C=64 etc... We didn't - I'm introducing the Memory Map Considerations here to show the example of what if we didn't think about how memory was going to be organized.

The C=64 only has 64k of memory of which typically the range $0800-$a000 is available and $c000-$cfff is also. If we had blindly picked numbers all over the place to store our code then we would have a disorganized program that would most likely accidentally use one subroutines storage as temporary data for another. It's like shooting randomly in Laser Tag not checking to see if there is a target there or not first... The end result: Chaos.

Currently we're not following the rule for "temporary variables" but as we gradually fade out of the normal C-64/128 default mode and write our own routines / interrupt handlers we'll switch things over. Also, on the C=128 instead of using Bank 0 with the I/O block enabled we're currently using the BANK 15 configuration as the program doesn't extend past $4000 yet ($0000-$4000 is common memory in the normal C=128 configuartion).

64 Considerations

The 64 will have free memory in the following areas: $0800-$a000, and $c000-$cfff. However, if we disable the Basic Rom we can have the whole area from $0800-$cfff free for our program. Because we don't need the Basic Rom we will do just that (in the listing now we currently won't but it will be done in a future issue). Therefore having the character set at $3000-$5000, the music data at $5000-$8000, the program will have the area free from $8000-$cfff. $0800-$3000 will be available if needed for routines who need temporary storage.

Temporary Storage is going to be defined as follows. Each routine that needs temporary storage will be assigned a "level" number. The lower levels will be assigned level 1 on up to level 3. The range $0800-$3000 will be broken down into the following sub-ranges.

```
     Level 1: $0800-$1000
     Level 2: $1000-$1800
     Level 3: $1800-$3000
```

This way when writing the sub-routines we can be assured that a section of memory is not overwritten by a subroutine we call. When we actually start programming we'll decide where in each sub-range the routine will have access to.

128 Considerations

The 128 has two "banks" of 64k each. Normally for large programs we would think about using both banks - (from the idea: Hey! - We got it, why not flaunt it?) but we won't be using both banks.

Free memory on the C128 typically consists of the range $0400-$09ff (where we'll be overwriting the 40 column screen (which we're gonna blank anyway) and the Basic run-time stack.) Also the area from $0b00-$0fff is free (overwriting the tape area, the rs-232 buffers,l and the sprite definition area). Also $1300-$cfff will be free.

Now, the C=128 has different memory maps it can configure itself to - Bank 15 is the standard mode under most basic programs and allows the programmer to directly "sys" to calls. The MMU (memory management unit - the chip that does everything) sees memory in a slightly differant way than from basic. We'll cover it in more detail when we examine the mem_init routine. For now, we're just gonna set up in the program and not in the coding segments. The explanation of what we're doing will be "revealed" in a future issue.

We will use Bank 0 of memory and from $1300+ will be the program. The ranges of $0400-$09ff and $0b00-$0fff will be used in a similair mannar as the C64 ranges were for Temporary Storage. We will also have the I/O section from $d000-$dfff swapped in. This is not a standard "basic BANK #" but when we cover the init'memory routine we'll see how we can do this. Music data will be from $a000-$d000.

X. Looking Forward / Back

Hopefully through the listing and the discussion of the routines you have started to understand the basic concept of programming: breaking down problems into smaller solvable steps. Try looking back over the code asking yourself why that instruction is there. What would happen if you switched the order? Is there an easier, better way to do the same thing? Why? Better yet, how? Examine the code, mess with it, muck it up so it doesn't work and then figure out exactly why. The only way to learn is by experimentation. (BTW, muck up a _copy_ of it - not the original ... *grins*)

Take a look at the different sections of code and analyze them to see how they do what they do. Take a look at how the code was organized in terms of simplification. Trace through each subroutine so that you're able to know what the return values will be. In other words: Study, Study, Study!! I'm in school and so I know I just used the dreaded 'S' word but that's what you're going to have to do if you're interested in learning 65xx/85xx machine language. The only way to learn it (easily) is to study other people's code and try to understand why they did what they did.

Next time we will take a look at the input routines for the mouse, joystick and keyboard scanning. In addition we will also allow the player to move the ship around on the screen to test the input drivers.

In addation, I am still looking for an individual to help with music and sound composition for this program. A knowledge of the SID chip and programming is helpful but not required. If you're willing to help then please email me at duck@pembvax1.pembroke.edu

XI. Listings

Because of the enormity of the program listing (some 1,500+ lines) it will not be listed in this article but will instead be available via anonymous ftp at ccosun.caltech.edu under pub/rknop/HACKING.MAG as invas1.sfx.

For those of you on the mailing list who would like to recieve it, a Mail-Server will be set up soon to handle requests and information will be sent to you concerning information about using it as soon as it's completed.

In the invasion1.sfx file there are the following files:

```
invasion.src - the main file
graphics.src - handles all graphics routines
logo.dat     - custom character logo
chars64.dat  - alien custom characters for C=64
chars128.dat - alien custom characters for C=128
titletxt.dat - text data for title screen
titlecol.dat - color data for title screen
invasion-128 - executable version of Space Invasion so far for C=128
invasion-64  - executable version of Space Invasion so far for C=64
```

Note: For the Commodore 128 it's recommended that you do a run/stop-restore and then a "BANK15:SYS7168" to execute the program. For the Commodore 64 it's recommended the border be changed via: "POKE53280,0:POKE53281,0:SYS 32768" to run the program.

==========

The Demo Corner:

FLI - more color to the screen
by Pasi 'Albert' Ojala (po87553@cs.tut.fi or albert@cc.tut.fi)
Written on 16-May-91 Translation 01-Jun-92

(All timings are in PAL, altho the principles will apply to NTSC too)

All of us have heard complaints about the color constraints on C64. One 8x8 pixel character position may only carry four different colors. FLI picture can have all of the 16 colors in one char position. What then is this FLI and how it is done ?

In the normal multicolor mode can one character position (4x8 pixels) have only four different colors and one of them is the common background color. Color codes are stored in half bytes (nybbles) to the video matrix memory (anywhere video matrix pointer points at, normally $0400) and to the color memory ($D800-$DBFF). In multicolor mode the color of each pixel is determined by two bits in the graphics memory. Bit pair 11 will refer to color memory, background color is the color for bit pair 00, and video matrix will define the colors for bit pairs 01 and 10.

What happens in the VIC ?

VIC (Video Interface Controller) fetches color information from memory on each bad line. This will steal time from processor, because VIC needs to use processor's bus cycles. Bad line is a curse in the C64 world. Fortunately VIC's data bus is 12 bits wide and so the color data fetch for each character position will take only one bus cycle. Color memory is physically wired to the VIC databus lines D8-D11.

How does VIC know where to fetch the graphical information ? Some of you know the mystical formulas needed to mess with the pixels in the hires screen. How are these functions obtained ? Are they just magic ? No, there are some internal counters in VIC. They always point to the right place in grafix memory and the address is determined like this:

```
A13  A12 A11 A10 A9  A8  A7  A6  A5  A4  A3  A2  A1  A0
CB13 VC9 VC8 VC7 VC6 VC5 VC4 VC3 VC2 VC1 VC0 RC2 RC1 RC0
```

Address bits A15 and A14 change according to the selected video bank.

Address bit A13 is CB13, which may be found in VIC register $18. It selects the right side of the video bank to be the bitmap memory. With these bits you can set the bitmap to eight different places in memory. However, some of them are useless because of the character ROM images and zero page/stack. Rest of the bits come from the internal counters.

VC9-VC0 (Video Counter) forms the address bits 12-3. The counter rolls through all 1000 character positions, 0-39 on the first eight lines, 40-79 on the second eight lines and so on. The lowest three bits come from the row counter, RC2-RC0. This is another VIC counter and it counts the scan lines from zero to seven.

A software graphics mode - FLI

VIC will systematically go through every byte in the bitmap memory, but how does it know where and when to get the color information ? This is where the main principle of FLI (Flexible Line Interpretation) lies. Color data is fetched (and this means it is a bad line), when the line counter matches with the vertical scroll register. VC9-VC0 defines where the color data is inside the video matrix and color memory.

If we change the vertical scroll register, we can fool VIC to think that every line is a bad line, so it will fetch the color information on every line too. Because VIC will fetch the colors continuosly, we can get independent colors on each scan line. We just have to change colors and VIC will handle the rest. Unfortunately the result is the loss of 40 processor cycles per line (see the Missing Cycles article for more information about VIC stealing cycles).

Doing it in practice

In practice there is no time to change color memory, but in multicolor mode VIC uses video matrix for color information too. We have just enough time to change the video matrix pointer, $D018. Now VIC will see a different video matrix on each scan line, different block of memory. With the four upper bits in the register we select one of the 16 video memories in the video bank. Just remember that the register also selects the position of the graphics memory (bitmap) inside the video bank.

Because we have to keep the bitmap in the same video bank, we only have half of the bank free for video matrices. Fortunately, that's all we need to get individual multicolor colors for each line and character position.
VIC will fetch the color data from the eight video matrices and then it will roll on to the next 40 bytes. After eight lines and matrices we will select the first video matrix again. (See picture 1)

Usually it is not necassary to use the whole screen for a FLI picture, especially if you want to have a scroller or some other effects. You just have to make sure that VIC is foolable in the usual way. The timing is also very important, even one cycle variations in the routine entry are not allowed. There is many ways to do the synchronization. One way is to use a sprite, as in the previous article. (See C= Hacking, Vol. 1, Iss. 3, The Demo Corner: Missing Cycles).

Not much time

Because a bad line will steal 40 cycles, there is only 23 cycles left on each scan line. It is enough for changing the video matrix and backgroud color. There is not a moment to lose, because you must change the vertical scroll register, video matrix pointer and the background color. This is why you can't have sprites in front of a FLI picture.

With FLI we get two selectable colors for each character position and line, each scan line can have it's own background color and each character position still has its own character color from color memory. In theory each character position could have 25 different colors, unfortunately VIC only has 16.

A little feature

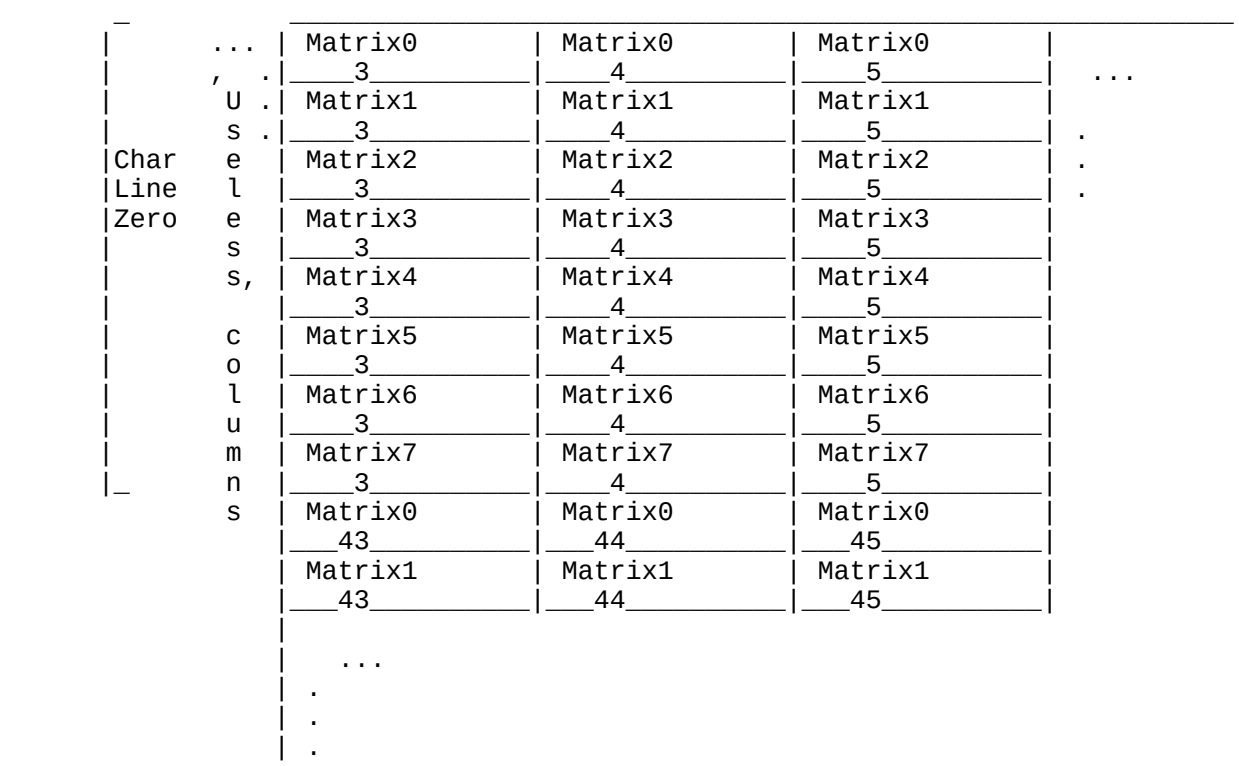
VIC does not like it when we change the vertical scroll register ($D011), and is a bit annoyed. It 'sees' code 255 (light gray) in video matrix and 9 (brown) in the color memory instead of the correct values stored there. Actually the color value seems to be the lower nybble of the data byte currently on the data bus (accessed by the processor (LDA#=$A9)). Unfortunately there is no chance to do the register change in the border and thus the three leftmost character columns are a bit useless, because the colors are fixed.

However, this doesn't mean that you can't use those three columns. FLI editors may not support the fixed colors though, so it may be hard to use them.

What to do with FLI ?

Because FLI will eat up all the available processor time (no Copper :-), it is not suitable for any action-games. Each FLI picture takes about 17 kB of memory: not so many pictures fit on one floppy. So, the only place for FLI is demos, intros, board-type games and maybe a GIF viewer..

Picture 1: From which matrix VIC fetches the multicolor values



Additional reading

If you have an Amiga you might want to get your hands into my conversion programs in C64GFX1.lha. The packet also includes FLI viewer for PAL C64's and some documentation about the FLI file format. It also has the same

utilities for Koala format pictures.

Available from:
cwaves.stfx.ca
nic.funet.fi:/pub/amiga/graphics/applications/convert

C64GFX.doc
                    C64Gfx1.0
          A C64 grafix format conversion package
)1991,1992 Pasi 'Albert' Ojala

          E-mail: po87553@cs.tut.fi
                  albert@cc.tut.fi

This package contains programs which are used to convert portable
pixmap (ppm) files to C64 graphics formats (FLI and koala) under
AmigaOS. The package includes C source codes for the programs, so
it is possible to port the programs to another environment. C64GFX1.1
includes Unix-compilable sources.

In addition to this package you need e.g. PBMPlus to convert Amiga
ilbm files to ppm first. And of course some way to transfer files
between the machines.

==
RS-232 Converter
by Warren Tustin (warren@col.hp.com)

  This article presents a way to interface from the C= rs232 hardware
behind the user port to a standard 25pin female rs232 connector using only
one IC and a few capacitors. It is not a UART or a SWIFTLINK type interface
which take place of the internal C= rs232 circuitry, but a simple level shifting
interface that uses the internal rs232 routines and translates the user port
levels to rs232 levels. Therefore you can only get upto 2400baud/9600baud
(C=64/C=128) with this design.

  The "old" way to do this was to used MC1488 and MC1489 parts (a line
driver and line receiver), however these required a negative supply to interface
properly. The user port only supplies +5volts, hence this presents a problem.
There has been success using these parts or discrete transistors and resistors
since many modems are somewhat friendly and seem to work even though the levels
were marginal. Also, some signals were not used, allowing for potential
problems. Another way to solve this problem was to buy a $25-30 interface. If
you can find the IC below, you have another choice that is relatively
inexpensive.

   The LT1133 is basically the MC1488 and 1489 put together into one part
with an internal charge pump scheme that allows the internal drivers to output
+5 and -5 volts to the rs232 connector. It also has enough drivers and
receivers to handle all of the signals that the C= uses for rs232.

   So with this IC, 5 capacitors and the two connectors (user port and
rs232) you can build your own interface to the standard 25 pin modem cable.

   Here are the plans for an User port TO RS232 connector using just ONE IC and
4 capacitors. It uses a Linear Technology LT1133 buffer that has 3 RS232
drivers and 5 receivers. It has worked for me with no problems and takes
a minimum amout of wiring to get to work. My board is only the width of
the user port and about 1.5 inches deep in size.

Parts list:

     LT1133CN plastic dip or LT1133CJ ceramic dip RS232 driver from Linear
        Technologies

(It takes 27mA max (17mA typical) so is well below the 100mA
limit of the user port)

```
Driver In pins  (15,19,21)  TTL/CMOS compatible.  Unused inputs
                  |  |  |   should be tied to +5v.
Driver Out pins (11, 7, 5)  RS232 compatible.  Short circuit
                            protected from -30v to +30v.
Receiver In pins  (6, 8, 9, 10,12)  Accept RS232 levels (+-30v)
                   |  |  |   |  |   and have 0.4v of hysteresis to
                   |  |  |   |  |   provide noise immunity.
Receiver Out pins (20,18,17,16,14)  TTL/CMOS outputs.
```

NOTE: Lines above indicate which inputs go with which outputs, and the pairs can be interchanged freely. I connected them as described below because the wiring worked out the best for me.

4 - >= 1uF capacitors Used to generate RS232 voltages by a charge pump technique inside IC

1 1uF capacitor To bypass the 5volt supply for noise rejection.

1 User port female connector. (I just dug this up, I'm not sure where these can be found, I think it is 0.159" spacing, 24pin.

1 RS232 25pin female connector. Can be found at R-Shack

Some sort of .1" spacing proto board

Connections:

User port connector (Looking into the C64 or C128)

```
 1  2  3  4  5  6  7  8  9 10 11 12
-----------------------------------
|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX|
-----------------------------------
 A  B  C  D  E  F  H  J  K  L  M  N
```

Ground & Power:

Pins 1, A, 12, N to Ground of board.
LT1133 pin 2 to pin 2 of User port connector (+5 volts)
1uF capacitor between pin2 and ground (bypass cap)
LT1133 pin 13 to Ground.
RS232 connector pins 7 & 1 to Ground.

LT1133 capacitors:

1uF from pin 1 (V+) to ground (If polarized (electrolytic) + side to pin 1)
1uF from pin 24 (V-) to ground (As above but + side to ground)
1uF from pin 3 (C1+) to pin 4 (C1-) (Again if polarized, + side to pin 3)
1uF from pin 22 (C2+) to pin 23 (C2-) (Again if polarized, + side to pin 22)

Commodore side				RS232 side	
User port pin	Signal name	Pin of LT1133	Signal direction	Pin of LT1133	RS232 connector
B	FLAG2	Din	20	<--	6	3
C	PB0	Din	also connect to above pin 20			
D	PB1	RTS	21	-->	5	4
E	PB2	DTR	19	<--	7	20
F	PB3	RI	18	-->	8	22
H	PB4	DCD	17	-->	9	8

```
   J   PB5      Not used
   K   PB6      CTS        16             -->         10            5
   L   PB7      DSR        14         <--             12            6
   M   PA2      Dout       15             -->         11            2
```

This assumes that you want to connect all of the communication lines. I did it this way because the C128 programmers reference guide had all of the signals above listed. If you want to drop RI (ring indicator) you could also use an LT1134 which has 4 drivers and 4 receivers.

Parts Substitution:

There are other IC's available which will work in this application. The rs232 bus levels for the LT parts are spec'ed at ~ +/- 7v typical, while the MAXIM parts are +/-9v typical. (Both are min at +/- 5v which should work in all applications). I think that an interface can be done with only 3 lines, Din(Rx), Dout(Tx), and either DSR or CTS, so if you can't get the LT1133 one of these others might work, although the pinouts would be different. The max you would need is what the LT1133 supplies, 3 drivers & 5 receivers.

Here is a description of parts that might be substituted:
(Note, with all of them you should use a bypass cap on the +5v supply which I have NOT included in the counts below.)

NAME	PINS	RS232 DRIVERS	RS232 RECEIVERS	# of EXTERNAL CAPS	COMMENTS
Linear Technology parts...					
(On the parts with shutdown (SD), the pin must be tied to +5 to operate)					
LT1133	24	3	5	4	In article above
LT1130	28	5	5	4	Overkill (SDp14)
LT1131	28	5	4	4	2 ex Dr, 1 < Rcvr
LT1132	24	5	3	4	2 ex Dr, 2 < Rcvr
LT1134	24	4	4	4	1 < Rcvr
LT1136	28	4	5	4	1 ex Dr (SDp14)
LT1137	28	3	5	4	SDp13
LT1138	28	5	3	4	LT1132 w/SDp13
MAXIM parts... (Some also have a TTL EN_ pin that must be tied to 0v to operate)					
(On these parts, SD must be tied to 0v to operate!)					
MAX232	16	2	2	4	May work
MAX233	20	2	2	0!	No external caps
MAX235	24	5	5	0!	Overkill, but NO caps, SDp21, EN_p20
MAX236	24	4	3	4	1 ex Dr, 2 < Rcvr, SDp21, EN_p20
MAX237	24	5	3	4	2 ex DR, 2 < Rcvr
MAX238	24	4	4	4	1 ex DR, 1 < Rcvr

In summary, you can see there are many different parts you could use, especially if you don't need all the signals. The MAXIM parts seem to do the job in fewer pins and a little better typical drive spec and I would recommend the MAX235 overall since you only need 1 bypass cap to make it operate!

==========

Introduction to the VIC-II
by Pasi 'Albert' Ojala (...)

The Video Interface Controller used in C64 have several different operating modes and different graphical primitives. Basically there is a) character mode, b) bitmap mode and c) movable objects that can be mixed with the other graphics. These primitives can also be put into a more colorful mode,

but you lose half of the resolution in that process.

I. Standard Character Display Mode

In the character display mode, VIC fetches character pointers from video matrix, which consists of 1000 8-bit bytes formatted as 25 rows of 40 characters each. The 8-bit character code implies 256 different characters simultaneously onscreen.

Each character code can have an unique image, which consists of 8 bytes in character memory. The position of the character memory can be moved with the character base pointer and thus it is possible to have several character sets simultaneosly in memory. One character memory is 2048 bytes.

In addition to the character code, each position in video matrix has an associated color nybble (4 bits) in color memory ($D800-$DFFF). For each zero-bit in the charset the background color from register $21 is displayed, the color-nybble is used for the one-bit.

II. Character Multicolor Mode

In character multicolor mode, color selection is increased. Each byte is fetched from the character memory just like in the standard character mode, but they are interpreted differently. In this mode bytes are divided into bit-pairs. For bit pair "00" the background color from register $21 is displayed, background color #1 is used for bit pair "01" and background color #2 for bit pair "10". The color nybble will define the color for bit pair "11".

The highest bit in the color memory defines whether the character is to be displayed in multicolor ("1") or in standard mode ("0"). Because of this, only colors in the range from 0 to 7 are possible for bit pair "11". And since two bits are required to specify one dot color, the character is now displayed as a 4x8 matrix instead of the 8x8 matrix and the size of the dots are doubled horizontally.

III. Extended Color Mode

The extended color mode allows the selection of one background color from four possibilities for each character position in the normal 8x8 resolution. The character image data is processed like in standard character mode, but the two most significant bits in the character code (video matrix) are used to select the right background color register. Only character images from 0 to 63 are accessible, because two of the most significant bits are used for the background color selection.

Extended color mode and multicolor mode should not be selected simultaneosly, because this will result a black screen. However, this is a very easy way to hide something if needed.

IV. Standard Bit Map Mode

In bit map mode, a one-to-one correspondence exists between each displayed dot and a memory bit. The bit map provides a resolution of 320H x 200V individually controlled pixels. The video matrix is still accessed as in character mode, but the data is interpreted as color data. When a bit is "0" in the bit map data, the color from the lower nybble is selected. The higher nybble from the video matrix is used for the bit "1".

V. Multicolor Bit Map Mode

In multicolor bit map mode two bits in the bit map memory determine the color of one pixel. If the bit pair is "11", the color found from the color memory is used. The background color is used for bit pair "00" and the video matrix defines the colors for bit pairs "01" and "10". As it takes two bits to define one pixel color, the horizaontal resolution is halved to 160H x 200V.

VI. Movable Object Blocks (MOBs)

The movable object block is a special type of graphical object which can be displayed independently from the other graphics. Each one of the MOBs can be moved independently anywhere in the screen. Eight unique MOBs can be displayed simulataniously, each defined by 64 bytes in memory which are displayed as a 24 x 21 pixel array.

Each MOB can be selectively enabled (MnE="1") or disabled (MnE="0"). A MOB is positioned via its X and Y position registers. Nine bits are needed to define the vertical position and the most significant bits of all MOBs are stored in the register $10. As X locations 23 to 347 and Y locations 50 to 249 are entirely visible on the screen, MOBs can be smoothly moved to an off-screen position.

Each MOB has its own color register and a MOB can be displayed either in standard or multicolor mode (MnMC="1"). As usually, multicolor mode gives more colors, but halves the horizontal resolution. In multicolor mode bit pair "00" is transparent, the MOB color register defines the color for pair "10", and MOB multicolor registers give the colors for pairs "01" and "11".

MOBs can be selectively expanded in both directions. When MOB is expanded, the pixel size also expands and it is still displayed as 24 x 21 matrix (12 x 21 in multicolor mode).

MOB priorities define whether a MOB appears behind or on top of the character or bit map graphics. A "1" in MnDP means MOB is displayed behind. MOB collision registers may be used to detect if a non-transparent data of two MOBs or a MOB and character or bitmap foreground data is colliding.

[Ed's Note: MOB's are Sprites. Commodore initially referred to them as MOB's and still does in some areas.]

VII. Other features

The display screen may be blanked by setting the DEN bit to a "0". The entire screen will be filled with the border color as set in register 32 ($20). When the screen is blanked, VIC will need only transparent memory cycles and the processor is not slowed down. However, MOB data is still fetched, if the MOBs are not also disabled.

The normal display consists of 25 rows of 40 characters each. The display window can be reduced to 24 rows and 38 characters. This has no effect on how the data is interpreted, only the characters next to the border are covered by the border. RSEL controls the number of rows ("1" for 25 rows) and CSEL controls the number of columns ("1" for 40 columns).

The display data may be scrolled up to one character space in both vertical and horizontal direction. Position of the screen is set with the 3 lowest order (least significant) bits in registers 22 ($16) and 17 ($11).

Light pen latch is used to catch the position of the light pen when a pulse is received in the LP pin. The value is latched only once in a frame.

The raster register is a dual-function register. A read from the raster register returns the current raster position and a write to it will set the

raster compare value. When the written value and the current raster line matches, a raster interrupt is generated if enabled. Raster register has its most significant (9th) bit in register 17 ($11).

The interrupt register shows the status of the four sources of interrupt. A corresponding bit will be set to "1" when an interrupt source has generated an interrupt request. To enable an interrupt request to set the /IRQ output to zero, the corresponding enable bit in register 26 ($1a) must be set to "1". The interrupt latch may only be cleared by writing a "1" to the desired latch in the interrupt register.

VIC register map (Base address $d000)

Address	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description
00 $00	M0X7	M0X6	M0X5	M0X4	M0X3	M0X2	M0X1	M0X0	MOB 0 X-position
01 $01	M0Y7	M0Y6	M0Y5	M0Y4	M0Y3	M0Y2	M0Y1	M0Y0	MOB 0 Y-position
02 $02	M1X7	M1X6	M1X5	M1X4	M1X3	M1X2	M1X1	M1X0	MOB 1 X-position
03 $03	M1Y7	M1Y6	M1Y5	M1Y4	M1Y3	M1Y2	M1Y1	M1Y0	MOB 1 Y-position
04 $04	M2X7	M2X6	M2X5	M2X4	M2X3	M2X2	M2X1	M2X0	MOB 2 X-position
05 $05	M2Y7	M2Y6	M2Y5	M2Y4	M2Y3	M2Y2	M2Y1	M2Y0	MOB 2 Y-position
06 $06	M3X7	M3X6	M3X5	M3X4	M3X3	M3X2	M3X1	M3X0	MOB 3 X-position
07 $07	M3Y7	M3Y6	M3Y5	M3Y4	M3Y3	M3Y2	M3Y1	M3Y0	MOB 3 Y-position
08 $08	M4X7	M4X6	M4X5	M4X4	M4X3	M4X2	M4X1	M4X0	MOB 4 X-position
09 $09	M4Y7	M4Y6	M4Y5	M4Y4	M4Y3	M4Y2	M4Y1	M4Y0	MOB 4 Y-position
10 $0a	M5X7	M5X6	M5X5	M5X4	M5X3	M5X2	M5X1	M5X0	MOB 5 X-position
11 $0b	M5Y7	M5Y6	M5Y5	M5Y4	M5Y3	M5Y2	M5Y1	M5Y0	MOB 5 Y-position
12 $0c	M6X7	M6X6	M6X5	M6X4	M6X3	M6X2	M6X1	M6X0	MOB 6 X-position
13 $0d	M6Y7	M6Y6	M6Y5	M6Y4	M6Y3	M6Y2	M6Y1	M6Y0	MOB 6 Y-position
14 $0e	M7X7	M7X6	M7X5	M7X4	M7X3	M7X2	M7X1	M7X0	MOB 7 X-position
15 $0f	M7Y7	M7Y6	M7Y5	M7Y4	M7Y3	M7Y2	M7Y1	M7Y0	MOB 7 Y-position
16 $10	M7X8	M6X8	M5X8	M4X8	M3X8	M2X8	M1X8	M0X8	MSB of X-position
17 $11	RC8	ECM	BMM	DEN	RSEL	Y2	Y1	Y0	(See text)
18 $12	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	Raster register
19 $13	LPX8	LPX7	LPX6	LPX5	LPX4	LPX3	LPX2	LPX1	Light Pen X
20 $14	LPY7	LPY6	LPY5	LPY4	LPY3	LPY2	LPY1	LPY0	Light Pen Y
21 $15	M7E	M6E	M5E	M4E	M3E	M2E	M1E	M0E	MOB Enable
22 $16	-	-	RES	MCM	CSEL	X2	X1	X0	(See text)
23 $17	M7YE	M6YE	M5YE	M4YE	M3YE	M2YE	M1YE	M0YE	MOB Y-expand
24 $18	VM13	VM12	VM11	VM10	CB13	CB12	CB11	-	Memory Pointers
25 $19	IRQ	-	-	-	ILP	IMMC	IMBC	IRST	Interrupt Register
26 $1a	-	-	-	-	ELP	EMMC	EMBC	ERST	Enable Interrupt
27 $1b	M7DP	M6DP	M5DP	M4DP	M3DP	M2DP	M1DP	M0DP	MOB-DATA Priority
28 $1c	M7MC	M6MC	M5MC	M4MC	M3MC	M2MC	M1MC	M0MC	MOB Multicolor select
29 $1d	M7XE	M6XE	M5XE	M4XE	M3XE	M2XE	M1XE	M0XE	MOB X-Expand
30 $1e	M7M	M6M	M5M	M4M	M3M	M2M	M1M	M0M	MOB-MOB Collision
31 $1f	M7D	M6D	M5D	M4D	M3D	M2D	M1D	M0D	MOB-DATA Collision
32 $20	-	-	-	-	EC3	EC2	EC1	EC0	Exterior Color
33 $21	-	-	-	-	B0C3	B0C2	B0C1	B0C0	Background #0 Color
34 $22	-	-	-	-	B1C3	B1C2	B1C1	B1C0	Background #1 Color
35 $23	-	-	-	-	B2C3	B2C2	B2C1	B2C0	Background #2 Color
36 $24	-	-	-	-	B3C3	B3C2	B3C1	B3C0	Background #3 Color
37 $25	-	-	-	-	MM03	MM02	MM01	MM00	MOB Multicolor #0
38 $26	-	-	-	-	MM13	MM12	MM11	MM10	MOB Multicolor #1
39 $27	-	-	-	-	M0C3	M0C2	M0C1	M0C0	MOB 0 Color
40 $28	-	-	-	-	M1C3	M1C2	M1C1	M1C0	MOB 1 Color
41 $29	-	-	-	-	M2C3	M2C2	M2C1	M2C0	MOB 2 Color
42 $2a	-	-	-	-	M3C3	M3C2	M3C1	M3C0	MOB 3 Color
43 $2b	-	-	-	-	M4C3	M4C2	M4C1	M4C0	MOB 4 Color
44 $2c	-	-	-	-	M5C3	M5C2	M5C1	M5C0	MOB 5 Color
45 $2d	-	-	-	-	M6C3	M6C2	M6C1	M6C0	MOB 6 Color

```
46 $2e  -     -     -     -     M7C3  M7C2  M7C1  M7C0  MOB 7 Color

MnX  = MOB n X position         MnY  = MOB n Y position
RC   = Raster compare register  ECM  = Extended color mode
MBB  = Bit map mode             DEN  = Display enable
RSEL = Row select               Y    = Screen Y position
LPX  = Light pen X position     LPY  = Light pen Y position
MnE  = MOB n Enable             RES  = Always set to zero!
MCM  = Multicolor mode          CSEL = Column select
X    = Screen X position        MnYE = MOB n Y expand
VM   = Video matrix pointer     CB   = Character base pointer
MnDP = MOB to data priority     MnMC = MOB n multicolor select
MnXE = MOB n X expand
```

==========

LITTLE RED READER: MS-DOS file reader for the 128 and 1571/81 drives.

by Craig Bruce <csbruce@neumann.uwaterloo.ca>

1. INTRODUCTION

This article presents a program that reads MS-DOS files and the root directory of MS-DOS disks. The program copies only from drive to drive without buffering file data internally. This is simpler and imposes no limits on the size of the files transferred, although it requires the use of two disk drives (or a logical drive). The user-interface code is written in BASIC and presents a full-screen file selection menu. The grunt-work code is written in assembly language and operates at maximum velosity.

The Burst Command Instruction Set of the 1571/81 is used to read the MS-DOS disk blocks and the standard kernel routines are used for outputting the data. (I am an operating systems specialist, so I call it a kernEl!) Thus, the MS-DOS files must be read from a 1571 or 1581 disk drive, but the output device may be any disk drive type, the screen or a printer, or a virtual drive type such as RAMLink, RAMDrive, or RAMDOS (for the REU). It is interesting to note that the data can be read in from an MS-DOS disk faster than it can be written out to a 1571, 1581, or even a RAMDOS file. A RAMLink can swallow the data only slightly faster than it can be read.

Little Red Reader (LRR) supports double density 3.5" disks formatted with 80 tracks, 9 sectors per track, and 2 sides with a 1581 and 5.25" double density disks formatted with 40 tracks, 9 sectors per track, and 2 sides with a 1571. A limit of 128 directory entries and 3 File Allocation Table (FAT) sectors is imposed. There must be 2 copies of the FAT and the cluster size may be 1 or 2 sectors. The sector size must be 512 bytes.

Oh, about the name. It is a play on the name of another MS-DOS file copier available for the C-128. "Little" means that it is smaller in scope than the other program, and "Red" is a different primary color to avoid any legal complications. It is also the non-white color of the flag of the country of origin of this program (no, I am not Japanese). Also, this program is Public Domain Software, as is all software I develop for 8-bit Commodore Computers. Feel free to E-mail me if you have questions or comments about this article.

2. USER GUIDE

LOAD and RUN the "lrr.128" BASIC program file. When the program is first run, it will display an "initializing" message and will load in the binary machine language package from the "current" Commodore DOS drive (the current drive is obtained from PEEK(186) - the last device accessed). The binary package is loaded only on the first run and is not reloaded on subsequent runs if the package ID field is in place.

2.1. MAIN SCREEN

The main screen of the program is then displayed. The main screen of the program will look something like this:

```
MS-DEV=9    MS-TYPE=1581    CBM-DEV=8

NUM  S  TRN  TYP  FILENAME  EXT  LENGTH
---  -  ---  ---  --------  ---  ------
  1  *  ASC  SEQ  HACK4     TXT  120732
  2     BIN  PRG  RAMDOS    SFX   34923

D=DIRECTORY  M=MS-DEV  F=CBM-DEV Q=QUIT
T=TOGGLE-COLUMN, C=COPY-FILES, +/- PAGE
```

except that immediately after starting up, "<no files>" will be displayed rather than filenames. The "MS-DEV" and "MS-TYPE" fields give the device number and type of the drive containing the MS-DOS disk to copy from, and the "CBM-DEV" gives the device number of the drive/virtual drive/character device to copy file data to.

Information about all MS-DOS files in the root directory of the MS-DOS disk is displayed in columns below the drive information. "NUM" gives the number of the MS-DOS file in the directory listing, and "S" indicates whether the file is "selected" or not. If the file is selected, an asterisk (*) is displayed; otherwise, a blank is displayed. When you later enter Copy Mode, only the files that have been "selected" are copied.

The "TRN" field indicates the character translation scheme to be used when the file is copied. A value of "BIN" (binary) means no translation and a value of "ASC" (ascii) means the file characters are to be translated from MS-DOS ASCII (or "ASCII-CrLf") to PETSCII. The "TYP" field indicates the type of Commodore-DOS file to create for writing the MS-DOS file contents into. The possible values are "SEQ" (sequential) and "PRG" (program). The values of the TRN and TYP fileds are set independently, so you can copy binary data to SEQ files and ascii data to PRG files if you wish.

The "FILENAME" and "EXT" fields give the filename and extension type of the MS-DOS files and "LENGTH" gives the exact length of the files in bytes. Note that if you perform "ASC" translation on a file, its PETSCII version will have a shorter length.

2.2. USER COMMANDS

The bottom of the screen gives the command summary. After starting the program, you will want to setup the MS-DOS and CBM-DOS drives with the "M" and "F" commands. Simply press the (letter) key corresponding to the command name to activate the command. Pressing M will prompt you for the MS-DOS Drive Number and the MS-DOS Drive Type. In both cases, type the number and press RETURN. (Sorry for insulting all non-novices out there, but I want to be complete). The MS-DOS drive number cannot be the same as the CBM-DOS drive number (since the program copies from drive-to-drive without internal buffering). For the drive type, enter an "8", "81", or "1581" for a 1581 drive or anything else for a 1571 drive.

Pressing F will prompt you for the CBM-DOS device number. You may enter a number from 0 to 30, except that it must not be the MS-DOS drive number. Enter a "1" for Cassette Drive (God forbid!), a "3" for the screen, a "4" for the printer (with an automatic secondary address of 7 (lowercase)), any number above 7 for a Commodore disk drive or special virtual drive, or a value of "0" for the special "null" drive. A CBM-DEV value of 0 will case the program to read MS-DOS files and do nothing with the output. You can use this feature to check out the raw reading speed of the program.

After setting up the drives, press D to read in the root directory off the MS-DOS disk. The data will come blazing in from the disk but BASIC will take its good ole time sifting through it. Filenames are displayed on the screen

as they are scanned in. The program will (eventually) return to the main screen and display the formatted file information. One note: the process of logging in a 1581 MS-DOS disk takes about 12 seconds (on my 1581, anyway), so be patient. An MS-DOS disk will have to be "logged in" every time you change MS-DOS disks. (Disks are logged in automatically).

A couple of notes about accessing MS-DOS disks: don't try to access a device that is not present because the machine language routines cannot handle this error for some reason and will lock up, requiring a STOP+RESTORE. Also, make sure that an actual MS-DOS disk is loaded into the drive. If you accidentally place Commodore-DOS disk into the MS-DOS drive, the 1581 will report an invalid boot parameters error (#60), but a 1571 will lock up (since I don't check the sector size and my burst routines are expecting 512 bytes to come out of a sector whereas Commodore disks have only 256 bytes per sector).

Now you are ready to pick what files you want copied and how you want them copied. You will notice that a "cursor" appears in the "S" column of the first file. You may move the cursor around with the cursor keys: UP, DOWN, LEFT, RIGHT, HOME, and CLR. CLR (SHIFT-HOME) will move the cursor back to the first file on the first screen. You can move the cursor among the select, translation, and file-type columns of all the files. Pressing a SPACE or a RETURN will toggle the value of the field that the cursor is on. To toggle all of the values of the "cursor" column (including files on all other screens), press T. You will notice that moving the cursor around and toggling fields is a bit sluggish, especially if you are in Slow mode on the 40-column screen. Did I mention that this program will run on either the 40 or 80-column screen? Toggling an entire column can take a couple of seconds.

If there are more than 18 MS-DOS files, you can press the "+" and "-" keys to move among all of the screens of files. The cursor movement keys will wrap around on the current screen. "+" is page forward, and "-" is page backward. The screens wrap around too.

After you have selected all of the files you want to copy and their translation and file-type fields have been set, press the C key to go into Copy Mode (next section). After copying, you are returned to the main screen with all of the field settings still intact. To exit from the program, press Q.

2.3. COPY MODE

When you enter copy mode, the screen will clear and the name of each selected file is displayed as it is being copied. If an error is encountered on either the MS-DOS or CBM-DOS drive during copying, an error message will be displayed and copying will continue (after you press a key for MS-DOS errors).

To generate a CBM-DOS filename from an MS-DOS filename, the eight filename characters are taken (including spaces) and a dot (.) and the three characters of the extension are appended. Then, all spaces are removed, and if the name ends with a dot (.) character, then that dot character is removed as well. I think this is fairly reasonable.

If there already is a file with the same filename on the CBM-DOS disk, then you will be prompted if you want to overwrite the file or not. Entering an "n" will abort the copying of that file and go on to the next file, and entering a "y" (or anything else) will cause the CBM-DOS file to be "scratched" and then re-written.

The physical copying of the file is done completely in machine language and nothing is displayed on the screen while this is happening, but you can follow things by looking at das blinkin lichtes and listening for clicks and grinds. You will probably be surprised by the MS-DOS file reading speed (I mean in a good way). The disk data is read in whole tracks and cached in memory and the directory information and the FAT are retained in memory as well. The result is that minimal time is spent reading disk data, and no costly seeks are required for opening a new MS-DOS file. A result is that small files are

copied one after another very quickly. You will have to wait, however, on the relatively slow standard kernel/Commodore-DOS file writing.

A few changes had to be made to the program to accomodate the RAMDOS program. RAMDOS uses memory from $2300 to $3FFF of RAM0, which is not really a good place for a device driver, and it uses some of the zero-page locations that I wanted to use. But, difficulties were overcome. The importance of RAMDOS compatibility is that if you only have one disk drive but you have an REU, you can use RAMDOS to store the MS-DOS files temporarily. If you only have one disk drive and no REU, you are SOL (Out of Luck) unless you can get a RamDisk-type program for an unexpanded 128. The RAMDOS program is available from FTP site "ccosun.caltech.edu" in file "/pub/rknop/util128/ramdosii.sfx". One note I found out about RAMDOS: you cannot use a

```
DOPEN#1,(CF$),U(CD),W
```

with it like you are supposed to be able to; you have to use a

```
DOPEN#1,(CF$+",W"),U(CD)
```

Here is a table of copying speeds for copying from 1571s and 1581s with ASC and BIN translation modes. All figures are in bytes/second. These results were obtained from copying a 127,280 byte text file (the text of C= Hacking Issue #3).

FROM \ TO:	"null"	RAMLink	RAMDOS	JD1581	JD1571
81-bin	5772	3441	2146	n/a	644
81-asc	5772	3434	2164	n/a	661
71-bin	4323	2991	1949	1821	n/a
71-asc	4323	2982	1962	1847	n/a

The "null" device is that "0" CBM-DOS device number, and a couple of entries are "n/a" since I only have one 1571 and one 1581. Note that my 71 and 81 are JiffyDOS-ified, so the performance of a stock 71/81 will be poorer. JiffyDOS gives about a 2x performance improvement for the standard file accessing calls (open, close, chrin, chrout). RAMDOS doesn't seem to be as snappy as you might think.

The "null" figures are quite impressive, but the raw sector reading speed without the overhead of mucking around with file organization is 6700 bytes/sec for a 1581 and 4600 B/s for a 71. The reason that the 1571 operates so quickly is that I use a sector interleave of 4 (which is optimal) for reading the tracks. I think that other MS-DOS file copier program uses an interleave of 1 (which is not optimal). I lose some of the raw performance because I copy the file data internally once before outputting it (to simplify some of the code).

In a couple of places you will notice that ASC translation gives slightly better or slightly worse performance than BIN. This is because although slightly more work is required to translate the characters, slightly fewer characters will have to be written to the CBM-DOS file, since PETSCII uses only CR where MS-DOS ASCII uses CR and LF to represent end-of-line. Translation is done by using a table (that you can change if you wish). Many entries in this table contain a value of zero, which means that no character will be output on translation. Most of the control characters and all of the characters of value 128 (0x80) or greater are thrown away on being translated. The table is set up so that CR characters are thrown away and the LF character is translated to a CBM-DOS CR character. Thus, both MS-DOS ASCII files and UNIX ASCII files can be translated correctly.

2. BURST COMMANDS

Three burst commands from the 1571/81 disk drive Burst Command Instruction Set are required to allow this program to read the MS-DOS disks: Query Disk

Format, Sector Interleave, and Read. The grungy details about issuing burst commands and burst mode handshaking are covered in C= Hacking Issue #3. The Query Disk Format command is used to "log in" the MS-DOS disk. The Inquire Disk burst command cannot be used with an MS-DOS disk on the 1581 for some unknown reason. I found this out the hard way. The Query Disk Format command has the following format:

BYTE \ bit:	7	6	5	4	3	2	1	0	Value
0	0	1	0	1	0	1	0	1	"U"
1	0	0	1	1	0	0	0	0	"0"
2	F	X	X	S	1	0	1	N	10

where the F, S, and N bits have a value of 0 for our purposes. A response of a burst status byte and six other throw-away bytes is given from the drive. This command takes quite a long time to execute on my 1581 but works quite quickly on my 1571. You only have to log in a disk whenever you change disks.

The Sector Interleave command is used to set a soft interleave for the Read command. I use an interleave of 1 for the 1581 and an interleave of 4 for the 1571. This means that the MS-DOS sectors will come from 1571 to the computer in the following order: 1, 5, 9, 4, 8, 3, 7, 2, 6 (there are 9 sectors per track on an MS-DOS disk (both 3.5" and 5.25"), numbered from 1 to 9). LRR handles the data coming in in this order, and in straight order from the 1581. The Sector Interleave command has the following format, where the W and N bits are 0 for us:

BYTE \ bit:	7	6	5	4	3	2	1	0	Value
0	0	1	0	1	0	1	0	1	"U"
1	0	0	1	1	0	0	0	0	"0"
2	W	X	X	0	1	0	0	N	8
3	<interleave>								1 or 4

The Read command is used to transfer the nine sectors of a track to the computer in the order specified by the interleave. The format is:

BYTE \ bit:	7	6	5	4	3	2	1	0	Value
0	0	1	0	1	0	1	0	1	"U"
1	0	0	1	1	0	0	0	0	"0"
2	T/L	E	B/X	S	0	0	0	N	0 or 16
3	<track>								???
4	<sector>								1
5	<number of sectors>								9

There are a couple of differences between the 1571 and 1581 versions of this command. Most important, the S bit (Side of disk to use) has the opposite meaning on the two drives. There's no good reason that I know of for this inconsistency. This is the reason that LRR needs to know what type of MS-DOS drive it is dealing with (plus interleaving).

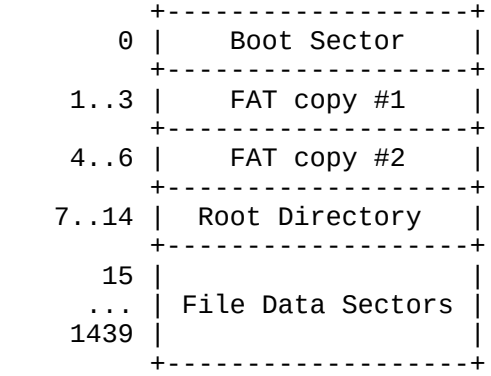
The read command returns the following data using burst mode handshaking:

0	Burst Status Byte
1	512 Data Bytes
...	
512	

for each sector transferred. If the Burst Status Byte indicates an error, then the data is not transferred and none of the following sectors are either. If the status byte gives a "Disk Changed" error, then you have to log in the disk with the Query Disk Format command before read will work properly. This is actually a good feature since it lets you know about a disk change so you can update any data structures you may have. LRR simply re-logs in the disk without updating any data structures and re-tries the failed read operation.

3. MS-DOS DISK FORMAT

An MS-DOS disk is separated into 4 different parts: the Boot Sector, the FAT(s), the Root Directory, and the File Data Sectors. The logical sectors (blocks) of a disk are numbered from 0 to some maximum number (1439 for a 3.5", 719 for a 5.25" DD disk). The physical layout and the logical sector numbers typically used by a 3.5" disk are shown here:



3.1. THE BOOT SECTOR

The Boot Sector is always at logical sector number 0. It contains some important information about the format of the disk and it also contains code to boot an MS-DOS machine from. We aren't concerned with the bootstrapping code, but the important values we need to obtain from the boot sector are:

ABBR	OFFSET	1571	1581	DESCRIPTION
CS	13	2	2	Cluster size in sectors
NB	14	1	1	Number of boot sectors
NF	16	2	2	Number of FATs
FL	23	2	3	FAT size in sectors
DE	17	112	112	Number of root directory entries
TS	19,20	720	1440	Total Number of sectors
NS	24	9	9	Number of sectors per track
NH	26	2	2	Number of sides

The 1571 and 1581 columns give the typical values of these parameters for 5.25" and 3.5" disks. The OFFSET is the address of the parameter within the boot sector. The total number of sectors is given in low-byte, high-byte order (since the 80x86 family is little-endian like the 6502 family). From the above parameters, we can derive the following important parameters:

ABBR	FORMULA	1571	1581	DESCRIPTION
F1	NB+NF*FL	5	7	First root directory sector
FS	NB+NF*FL+DE	12	14	First file data sector number
NC	(TS-FS)/CS	354	713	Total number of file clusters

LRR imposes a number of limits on these parameters and will error-out if you try to use a disk that is outside of LRR's limits.

3.2. CHEWING THE FAT

MS-DOS disks use a data structure called a File Allocation Table (FAT) to record which clusters belong to which file in what order and which blocks are free. A cluster is a set of contiguous sectors which are allocated to files as a group. LRR handles cluster sizes of 1 and 2 sectors, giving a logical file block size of 512 or 1024 bytes. Typically, a cluster size of 2 sectors is used.

The FAT is an array of 12-bit numbers, with an entry corresponding to each cluster that can be allocated to files. FAT entries 0 and 1 are reserved. If a FAT entry contains a value of $000, then the corresponding cluster is free and can be allocated to a file; otherwise, the cluster is allocated and the FAT entry contains the number of the NEXT FAT entry that belongs to the file. Thus, MS-DOS files are stored in a singly-linked list of clusters like Commodore-DOS files are, except that the links are not in the data sectors but rather are in the FAT. The pointer to the first FAT entry for a file is given in the file's directory entry.

A special NULL/NIL pointer value of $FFF is used to indicate the end of the chain of clusters allocated to a file. This value is stored in the FAT entry of the last cluster allocated to a file (of course). Consider the following example FAT:

```
   ENTRY       VALUE
   -----       -----
    $000        $FFF
    $001        $FFF
    $002   |----$003 <------Directory Entry
    $003   +--> $005----+
    $004        $000    |
    $005        $FFF <--+
```

Entries 0 and 1 are insignificant since they are reserved. Say that a file starts at FAT entry #2. Then, it consists of the following chain of clusters: 2, 3, and 5. Cluster #4 is free. Clusters can be allocated to a file in random order, but if they are allocated contiguously in forward order, then the file will be able to be read faster. The FAT is such an important data structure that typically two copies are kept on the disk incase one of them should become corrupted.

The MS-DOS designers were a little sneaky in storing the 12-bit FAT entries - they used only 12 real bits per entry. Ie., they store two FAT entries in three bytes, where the two entries share the two nybbles of the middle byte. The following diagram shows how the nybbles 1 (high), 2 (mid), and 3 (low) are stored into FAT entries A and B:

```
     BYTE:       0           1           2
              +---+---+   +---+---+   +---+---+
    ENTRY:    | A | A |   | B | A |   | B | B |
   NYBBLE:    | 2 | 3 |   | 3 | 1 |   | 1 | 2 |
              +---+---+   +---+---+   +---+---+
```

Anyway, let's just say it's a bit tricky to extract the 12-bit values from this compressed data structure. On top of that, I don't think there is any saving in disk space resulting from compressing this structure; they might as well have just used a 16-bit FAT (like they do nowadays on larger disks).

3.3. THE ROOT DIRECTORY

The root directory has a fixed size, although I don't think that subdirectories do. LRR cannot access subdirectories. Each 512-byte sector of the root directory contains sixteen 32-byte directory entries. One directory entry is required for each file stored on the disk. A directory entry has the following structure:

OFFSET	LEN	DESCRIPTION
0..7	8	Filename
8..10	3	Extension
11	1	<unused?>
12	1	Attributes: $10=Directory, $08=VolumeId
13..21	9	<unused>
22..25	4	Date
26..27	2	Starting FAT entry number
28..31	4	File length in bytes

The filename and extension are stored with trailing padding spaces. If a directory entry is unused or deleted, then the first character of the filename is either a $00 or a $E5 (229). This is why you have to provide the first character of a filename if you are undeleting a file on an MS-DOS machine. Note that there is enough unused space that Microsoft or IBM could have ditched the annoying 8+3 character filename format.

The attributes bits tell whether the directory entry is for a regular file, a subdirectory, a disk volume name (in which case there is no file data), and a couple of other things I can't remember. I'm not sure about the exact position or format of the date, but LRR doesn't use it anyway. The starting FAT entry number and the file length are stored in lower-byte-first order.

3.4. THE FILE DATA SPACE

The ramainder of the disk space is used for storing file data in clusters of 1 or 2 sectors each. Given a cluster number (which is also the FAT entry number), the following formula is used to calculate the starting logical sector number of the cluster:

```
(ClusterNumber - 2) * ClusterSizeInBlocks + FirstFileDataLogicalSectorNumber
```

where "FirstFileDataLogicalSectorNumber" is the "FS" parameter derived earlier. The following consecutive logical sector numbers up to the number of sectors per cluster form the rest of the cluster. Note that a single cluster can span sectors from one side of the disk to another or from one track to another. We perform the "(ClusterNumber - 2)" portion of the calculation since the first two FAT entries are reserved.

Since the Read burst command of the 1571/81 wants the side, track, and sector number of a sector rather than its logical number, we also need formulae for these conversions:

```
Track  = LogicalSectorNumber / 18
Sector = LogicalSectorNumber % 9 + 1
Side   = (LogicalSectorNumber / 9) % 2
```

These formulae are more problematic than the previous one since they require division by 9 and 18. LRR uses the method of repeated subtraction to perfrom the necessary division (only one division is necessary). The above formulae imply that sequential logical sectors are stored on the top of the disk first and then the bottom of the disk of the same track, and then on the top of the next track, etc. This is a good sector numbering scheme (unlike the CBM-DOS scheme for 1571 sectors) since it is faster to switch sides of the disk than it is to switch tracks, so you can read the disk faster.

Oh yeah, the way that you know how many file data bytes are in the last cluster of a file chain (the cluster with the NULL FAT entry) is to take the file length from the directory entry and "mod" (the C language % operator) it with the cluster size. One special case is if this calculation results in a zero, then the last cluster is completely full (rather than completely empty as the calculation would suggest). This calculation is easily done in machine language with an AND operation since the cluster size is always a

power of two.

4. FILE COPYING PACKAGE

This section discusses the interface to and implementation of the MS-DOS file copying package. It is written in assembly language and is loaded into memory at address $8000 on bank 0 and requires about 13K of memory. The package is loaded at this high address to be out of the way of the main BASIC program, even if RAMDOS is installed.

4.1. INTERFACE

The subroutine call and parameter passing interface to the file copying package is summarized as follows:

ADDRESS	DESCRIPTION
PK	InitPackage subroutine
PK+3	LoadDirectory subroutine
PK+6	CopyFile subroutine
PK+9	two-byte package identification number
PK+15	errno : error code returned
PK+16	MS-DOS device number (8 to 30)
PK+17	MS-DOS device type ($00=1571, $FF=1581)
PK+18	two-byte starting cluster number for file copying
PK+20	low and mid bytes of file length for copying

where "PK" is the load address of the package. Additional subroutine parameters are passed in the processor registers.

The "InitPackage" subroutine should be called when the package is first installed, whenever the MS-DOS device number is changed, and whenever a new disk is mounted to invalidate the internal track cache. It requires no parameters.

The "LoadDirectory" subroutine will load the directory, FAT, and the Boot Sector parameters into the internal memory of the package from the current MS-DOS device number. No (other) input parameters are needed and the subroutine returns a pointer to the directory space in the .AY registers and the number of directory entries in the .X register. If an error occurs, then the subroutine returns with the Carry flag set and the error code is available in the "errno" interface variable. The directory entry data is in the directory space as it was read in raw from the directory sectors on the MS-DOS disk.

The "CopyFile" subroutine will copy a single file from the MS-DOS disk to a specified CBM-Kernal logical file number (the CBM file must already be opened). If the CBM logical file number is zero, then the file data is simply discarded after it is read from the MS-DOS file. The starting cluster number of the file to copy and the low and mid bytes of the file length are passed in the PK+18 and PK+20 interface words. The translation mode to use is passed in the .A register ($00=binary, $FF=ascii) and the CBM logical file number to output to is passed in the .X register. If an error occurs, the routine returns with the Carry flag set and the error code in the "errno" interface variable. There are no other output parameters.

Note that since the starting cluster number and low-file length of the file to be copied are required rather than the filename, it is the responsibility of the front-end application program to dig through the raw directory sector data to get this information. The application must also open the Commodore-DOS file of whatever filetype on whatever device is required; the package does not need to know the Commodore-DOS device number.

The MS-DOS device number and device type interface variables allow you to set the MS-DOS drive and the package identification number allows the application

program to check if the package is already loaded into memory so that it only has to load the package the first time the application is run and not on re-runs. The identification sequence is a value of $CB followed by a value of 131.

4.2. IMPLEMENTATION

This section presents the code that implements the MS-DOS file reading package. It is here in a special form; each code line is preceded by a few special characters and the line number. The special characters are there to allow you to easily extract the assembler code from the rest of this magazine (and all of my ugly comments). On a Unix system, all you have to do is execute the following command line (substitute filenames as appropriate):

```
grep '^\.%...\!' Hack4 | sed 's/^.%...\!..//' | sed 's/.%...\!//' >lrr.s
```

You'll notice that the initial comment lines here were an afterthought.

```
.%000!  ;Little Red Reader MS-DOS file copier program
.%000!  ;written 92/10/03 by Craig Bruce for C= Hacking Net Magazine
.%000!
```

The code is written for the Buddy assembler and here are a couple setup directives. Note that my comments come before the section of code.

```
.%001!  .org $8000
.%002!  .obj "@:lrr.bin"
.%003!
.%004!  ;====jump table and parameters interface ====
.%005!
.%006!  jmp initPackage
.%007!  jmp loadDirectory
.%008!  jmp copyFile
.%009!
.%010!  .byte $cb,131   ;identification
.%011!  .byte 0,0,0,0
.%012!
```

These variables are included in the package program space to minimize unwanted interaction with other programs loaded at the same time, such as the RAMDOS device driver.

```
.%013!  errno           .buf 1    ;(location pk+15)
.%014!  sourceDevice    .buf 1
.%015!  sourceType      .buf 1    ;$00=1571, $ff=1581
.%016!  startCluster    .buf 2
.%017!  lenML           .buf 2    ;length medium and low bytes
.%018!
.%019!  ;====global declaraions====
.%020!
.%021!  kernelListen = $ffb1
.%022!  kernelSecond = $ff93
.%023!  kernelUnlsn  = $ffae
.%024!  kernelAcptr  = $ffa2
.%025!  kernelCiout  = $ffa8
.%026!  kernelSpinp  = $ff47
.%027!  kernelChkout = $ffc9
.%028!  kernelClrchn = $ffcc
.%029!  kernelChrout = $ffd2
.%030!
.%031!  st = $d0
.%032!  ciaClock = $dd00
.%033!  ciaFlags = $dc0d
.%034!  ciaData  = $dc0c
.%035!
```

```
These are the parameters and derived parameters from the boot sector.  They
are kept in the program space to avoid interactions.

.%036!  clusterBlockCount .buf 1        ;1 or 2
.%037!  fatBlocks         .buf 1        ;up to 3
.%038!  rootDirBlocks     .buf 1        ;up to 8
.%039!  rootDirEntries    .buf 1        ;up to 128
.%040!  totalSectors      .buf 2        ;up to 1440
.%041!  firstFileBlock    .buf 1
.%042!  firstRootDirBlock .buf 1
.%043!  fileClusterCount  .buf 2
.%044!

The cylinder (track) and side that is currently stored in the trach cache.

.%045!  bufCylinder     .buf 1
.%046!  bufSide         .buf 1
.%047!  formatParms     .buf 6
.%048!

This package is split into a number of levels.  This level interfaces with the
Kernal serial bus routines and the burst command protocol of the disk drives.

.%049!  ;====hardware level====
.%050!

Connect to the MS-DOS device and send the "U0" burst command prefix and the
burst command byte.

.%051!  sendU0 = *  ;( .A=burstCommandCode ) : .CS=err
.%052!     pha
.%053!     lda #0
.%054!     sta st
.%055!     lda sourceDevice
.%056!     jsr kernelListen
.%057!     lda #$6f
.%058!     jsr kernelSecond
.%059!     lda #"u"
.%060!     jsr kernelCiout
.%061!     bit st
.%062!     bmi sendU0Error
.%063!     lda #"0"
.%064!     jsr kernelCiout
.%065!     pla
.%066!     jsr kernelCiout
.%067!     bit st
.%068!     bmi sendU0Error
.%069!     clc
.%070!     rts
.%071!
.%072!     sendU0Error = *
.%073!     lda #5
.%074!     sta errno
.%075!     sec
.%076!     rts
.%077!

Toggle the "Data Accepted / Ready For More" clock signal for the burst
transfer protocol.

.%078!  toggleClock = *
.%079!     lda ciaClock
.%080!     eor #$10
.%081!     sta ciaClock
```

```
.%082!     rts
.%083!

Wait for a burst byte to arrive in the serial data register of CIA#1 from the
fast serial bus.

.%084!  serialWait = *
.%085!     lda #$08
.%086!  -  bit ciaFlags
.%087!     beq -
.%088!     rts
.%089!

Wait for and get a burst byte from the fast serial bus, and send the "Data
Accepted" signal.

.%090!  getBurstByte = *
.%091!     jsr serialWait
.%092!     ldx ciaData
.%093!     jsr toggleClock
.%094!     txa
.%095!     rts
.%096!

Send the burst commands to "log in" the MS-DOS disk and set the Read sector
interleave factor.

.%097!  mountDisk = *  ;() : .CS=err
.%098!     lda #%00011010
.%099!     jsr sendU0
.%100!     bcc +
.%101!     rts
.%102!  +  jsr kernelUnlsn
.%103!     bit st
.%104!     bmi sendU0Error
.%105!     clc
.%106!     jsr kernelSpinp
.%107!     bit ciaFlags
.%108!     jsr toggleClock
.%109!     jsr getBurstByte
.%110!     sta errno
.%111!     and #$0f
.%112!     cmp #2
.%113!     bcs mountExit

Grab the throw-away parameters from the mount operation.

.%114!     ldy #0
.%115!  -  jsr getBurstByte
.%116!     sta formatParms,y
.%117!     iny
.%118!     cpy #6
.%119!     bcc -
.%120!     clc

Set the sector interleave to 1 for a 1581 or 4 for a 1571.

.%121!     ;** set interleave
.%122!     lda #%00001000
.%123!     jsr sendU0
.%124!     bcc +
.%125!     rts
.%126!  +  lda #1            ;interleave of 1 for 1581
.%127!     bit sourceType
.%128!     bmi +
```

```
.%129!     lda #4            ;interleave of 4 for 1571
.%130!  +  jsr kernelCiout
.%131!     jsr kernelUnlsn
.%132!     mountExit = *
.%133!     rts
.%134!
```

Read all of the sectors of a given track into the track cache.

```
.%135!  bufptr = 2
.%136!  secnum = 4
.%137!
.%138!  readTrack = *  ;( .A=cylinder, .X=side ) : trackbuf, .CS=err
.%139!     pha
.%140!     txa
```

Get the side and put it into the command byte. Remember that we have to flip the side bit for a 1581.

```
.%141!     and #$01
.%142!     asl
.%143!     asl
.%144!     asl
.%145!     asl
.%146!     bit sourceType
.%147!     bpl +
.%148!     eor #$10
.%149!  +  jsr sendU0
.%150!     bcc +
.%151!     rts
.%152!  +  pla                  ;cylinder number
.%153!     jsr kernelCiout
.%154!     lda #1               ;start sector number
.%155!     jsr kernelCiout
.%156!     lda #9               ;sector count
.%157!     jsr kernelCiout
.%158!     jsr kernelUnlsn
```

Prepare to receive the track data.

```
.%159!     sei
.%160!     clc
.%161!     jsr kernelSpinp
.%162!     bit ciaFlags
.%163!     jsr toggleClock
.%164!     lda #<trackbuf
.%165!     ldy #>trackbuf
.%166!     sta bufptr
.%167!     sty bufptr+1
```

Get the sector data for each of the 9 sectors of the track.

```
.%168!     lda #0
.%169!     sta secnum
.%170!  -  bit sourceType
.%171!     bmi +
```

If we are dealing with a 1571, we have to set the buffer pointer for the next sector, taking into account the soft interleave of 4.

```
.%172!     jsr get1571BufPtr
.%173!  +  jsr readSector
.%174!     bcs trackExit
.%175!     inc secnum
.%176!     lda secnum
```

```
.%177!     cmp #9
.%178!     bcc -
.%179!     clc
.%180!     trackExit = *
.%181!     cli
.%182!     rts
.%183!

Get the buffer pointer for the next 1571 sector.

.%184!  get1571BufPtr = *
.%185!     lda #<trackbuf
.%186!     sta bufptr
.%187!     ldx secnum
.%188!     clc
.%189!     lda #>trackbuf
.%190!     adc bufptr1571,x
.%191!     sta bufptr+1
.%192!     rts
.%193!
.%194!  bufptr1571 = *
.%195!     .byte 0,8,16,6,14,4,12,2,10
.%196!

Read an individual sector into memory at the specified address.

.%197!  readSector = *  ;( bufptr ) : .CS=err

Get and check the burst status byte for errors.

.%198!     jsr getBurstByte
.%199!     sta errno
.%200!     and #$0f
.%201!     cmp #2
.%202!     bcc +
.%203!     rts
.%204!  +  ldx #2
.%205!     ldy #0
.%206!

Receive the 512 sector data bytes into memory.

.%207!     readByte = *
.%208!     lda #$08
.%209!  -  bit ciaFlags
.%210!     beq -
.%211!     lda ciaClock
.%212!     eor #$10
.%213!     sta ciaClock
.%214!     lda ciaData
.%215!     sta (bufptr),y
.%216!     iny
.%217!     bne readByte
.%218!     inc bufptr+1
.%219!     dex
.%220!     bne readByte
.%221!     rts
.%222!

This next level of routines deals with logical sectors and the track cache
rather than with hardware.

.%223!  ;====logical sector level====
.%224!
```

```
Invalidate the track cache if the MS-DOS drive number is changed or if a new
disk is inserted.  This routine has to establish a RAM configuration of $0E
since it will be called from RAM0.  Configuration $0E gives RAM0 from $0000 to
$BFFF, Kernal ROM from $C000 to $FFFF, and the I/O space over the Kernal from
$D000 to $DFFF.  This configuration is set by all application interface
subroutines.

.%225!  initPackage = *
.%226!     lda #$0e
.%227!     sta $ff00
.%228!     lda #$ff
.%229!     sta bufCylinder
.%230!     sta bufSide
.%231!     clc
.%232!     rts
.%233!

Locate a sector (block) in the track cache, or read the corresponding physical
track into the track cache if necessary.  This routine accepts the cylinder,
side, and sector numbers of the block.

.%234!  sectorSave = 5
.%235!
.%236!  readBlock = *  ;( .A=cylinder,.X=side,.Y=sector ) : .AY=blkPtr,.CS=err

Check if the correct track is in the track cache.

.%237!     cmp bufCylinder
.%238!     bne readBlockPhysical
.%239!     cpx bufSide
.%240!     bne readBlockPhysical

If so, then locate the sector's address and return that.

.%241!     dey
.%242!     tya
.%243!     asl
.%244!     clc
.%245!     adc #>trackbuf
.%246!     tay
.%247!     lda #<trackbuf
.%248!     clc
.%249!     rts
.%250!

Here, we have to read the physical track into the track cache.  We save the
input parameters and call the hardware-level track-reading routine.

.%251!     readBlockPhysical = *
.%252!     sta bufCylinder
.%253!     stx bufSide
.%254!     sty sectorSave
.%255!     jsr readTrack

Check for errors.

.%256!     bcc readBlockPhysicalOk
.%257!     lda errno
.%258!     and #$0f
.%259!     cmp #11    ;disk change
.%260!     beq +
.%261!     sec
.%262!     rts

If the error that happened is a "Disk Change" error, then mount the disk and
```

try to read the physical track again.

```
.%263!  +  jsr mountDisk
.%264!     lda bufCylinder
.%265!     ldx bufSide
.%266!     ldy sectorSave
.%267!     bcc readBlockPhysical
.%268!     rts
.%269!
```

Here, the physical track has been read into the track cache ok, so we recover the original input parameters and try the top of the routine again.

```
.%270!     readBlockPhysicalOk = *
.%271!     lda bufCylinder
.%272!     ldx bufSide
.%273!     ldy sectorSave
.%274!     jmp readBlock
.%275!
```

Divide the given number by 18. This is needed for the calculations to convert a logical sector number to the corresponding physical cylinder, side, and sector numbers that the lower-level routines require. The method of repeated subtraction is used. This routine would probably work faster if we tried to repeatedly subtract 360 (18*20) at the top, but I didn't bother.

```
.%276!  divideBy18 = *  ;( .AY=number ) : .A=quotient, .Y=remainder
.%277!     ;** could repeatedly subtract 360 here
.%278!     ldx #$ff
.%279!  -  inx
.%280!     sec
.%281!     sbc #18
.%282!     bcs -
.%283!     dey
.%284!     bpl -
.%285!     clc
.%286!     adc #18
.%287!     iny
.%288!     tay
.%289!     txa
.%290!     rts
.%291!
```

Convert the given logical block number to the corresponding physical cylinder, side, and sector numbers. This routine follows the formulae given earlier with a few simplifying tricks.

```
.%292!  convertLogicalBlockNum = *  ;( .AY=blockNum ) : .A=cyl, .X=side, .Y=sec
.%293!     jsr divideBy18
.%294!     ldx #0
.%295!     cpy #9
.%296!     bcc +
.%297!     pha
.%298!     tya
.%299!     sbc #9
.%300!     tay
.%301!     pla
.%302!     ldx #1
.%303!  +  iny
.%304!     rts
.%305!
```

Copy a sequential group of logical sectors into memory. This routine is used by the directory loading routine to load the FAT and Root Directory, and is used by the cluster reading routine to retrieve all of the blocks of a

cluster. After the given starting logical sector number is converted into its physical cylinder, side, and sector equivalent, the physical values are incremented to get the address of successive sectors of the group. This avoids the overhead of the logical to physical conversion. Quite a number of temporaries are needed.

```
.%306!  destPtr = 6
.%307!  curCylinder = 8
.%308!  curSide = 9
.%309!  curSector = 10
.%310!  blockCountdown = 11
.%311!  sourcePtr = 12
.%312!
.%313!  copyBlocks = *  ;( .AY=startBlock, .X=blockCount, ($6)=dest ) : .CS=err
.%314!     stx blockCountdown
.%315!     jsr convertLogicalBlockNum
.%316!     sta curCylinder
.%317!     stx curSide
.%318!     sty curSector
.%319!
.%320!     copyBlockLoop = *
.%321!     lda curCylinder
.%322!     ldx curSide
.%323!     ldy curSector
.%324!     jsr readBlock
.%325!     bcc +
.%326!     rts
.%327!  +  sta sourcePtr
.%328!     sty sourcePtr+1
.%329!     ldx #2
.%330!     ldy #0
```

Here I unroll the copying loop a little bit to cut the overhead of the branch instruction in half. (A cycle saved... you know).

```
.%331!  -  lda (sourcePtr),y
.%332!     sta (destPtr),y
.%333!     iny
.%334!     lda (sourcePtr),y
.%335!     sta (destPtr),y
.%336!     iny
.%337!     bne -
.%338!     inc sourcePtr+1
.%339!     inc destPtr+1
.%340!     dex
.%341!     bne -
```

Increment the cylinder, side, sector values.

```
.%342!     inc curSector
.%343!     lda curSector
.%344!     cmp #10
.%345!     bcc +
.%346!     lda #1
.%347!     sta curSector
.%348!     inc curSide
.%349!     lda curSide
.%350!     cmp #2
.%351!     bcc +
.%352!     lda #0
.%353!     sta curSide
.%354!     inc curCylinder
.%355!  +  dec blockCountdown
.%356!     bne copyBlockLoop
.%357!     clc
```

```
.%358!     rts
.%359!
```

Read a cluster into the Cluster Buffer, given the cluster number. The cluster number is converted to a logical sector number and then the sector copying routine is called. The formula given earlier is used for the conversion.

```
.%360!  readCluster = *  ;( .AY=clusterNumber ) : clusterBuf, .CS=err
.%361!     ;** convert cluster number to logical block number
.%362!     sec
.%363!     sbc #2
.%364!     bcs +
.%365!     dey
.%366!  +  ldx clusterBlockCount
.%367!     cpx #1
.%368!     beq +
.%369!     asl
.%370!     sty 7
.%371!     rol 7
.%372!     ldy 7
.%373!  +  clc
.%374!     adc firstFileBlock
.%375!     bcc +
.%376!     iny
.%377!
.%378!     ;** read logical blocks comprising cluster
.%379!  +  ldx #<clusterBuf
.%380!     stx 6
.%381!     ldx #>clusterBuf
.%382!     stx 7
.%383!     ldx clusterBlockCount
.%384!     jmp copyBlocks
.%385!
```

This next level of routines deal with the data structures of the MS-DOS disk format.

```
.%386!  ;====MS-DOS format level====
.%387!
.%388!  bootBlock = 2
.%389!
```

Read the disk format parameters, directory, and FAT into memory.

```
.%390!  loadDirectory = *  ;( ) : .AY=dirbuf, .X=dirEntries, .CS=err
.%391!     lda #$0e
.%392!     sta $ff00
.%393!
```

Read the boot sector and extract the parameters.

```
.%394!     ;** get parameters from boot sector
.%395!     lda #0
.%396!     ldy #0
.%397!     jsr convertLogicalBlockNum
.%398!     jsr readBlock
.%399!     bcc +
.%400!     rts
.%401!  +  sta bootBlock
.%402!     sty bootBlock+1
.%403!     ldy #13              ;get cluster size
.%404!     lda (bootBlock),y
.%405!     sta clusterBlockCount
.%406!     cmp #3
.%407!     bcc +
```

```
.%408!

If a disk parameter is found to exceed the limits of LRR, error code #60 is
returned.

.%409!     invalidParms = *
.%410!     lda #60
.%411!     sta errno
.%412!     sec
.%413!     rts
.%414!
.%415!  +  ldy #16              ;check FAT replication count, must be 2
.%416!     lda (bootBlock),y
.%417!     cmp #2
.%418!     bne invalidParms
.%419!     ldy #22              ;get FAT size in sectors
.%420!     lda (bootBlock),y
.%421!     sta fatBlocks
.%422!     cmp #4
.%423!     bcs invalidParms
.%424!     ldy #17              ;get directory size
.%425!     lda (bootBlock),y
.%426!     sta rootDirEntries
.%427!     cmp #129
.%428!     bcs invalidParms
.%429!     lsr
.%430!     lsr
.%431!     lsr
.%432!     lsr
.%433!     sta rootDirBlocks
.%434!     ldy #19              ;get total sector count
.%435!     lda (bootBlock),y
.%436!     sta totalSectors
.%437!     iny
.%438!     lda (bootBlock),y
.%439!     sta totalSectors+1
.%440!     ldy #24              ;check sectors per track, must be 9
.%441!     lda (bootBlock),y
.%442!     cmp #9
.%443!     bne invalidParms
.%444!     ldy #26
.%445!     lda (bootBlock),y
.%446!     cmp #2               ;check number of sides, must be 2
.%447!     bne invalidParms
.%448!     ldy #14              ;check number of boot sectors, must be 1
.%449!     lda (bootBlock),y
.%450!     cmp #1
.%451!     bne invalidParms
.%452!

Calculate the derived parameters.

.%453!     ;** get derived parameters
.%454!     lda fatBlocks        ;first root directory sector
.%455!     asl
.%456!     clc
.%457!     adc #1
.%458!     sta firstRootDirBlock
.%459!     clc                  ;first file sector
.%460!     adc rootDirBlocks
.%461!     sta firstFileBlock
.%462!     lda totalSectors     ;number of file clusters
.%463!     ldy totalSectors+1
.%464!     sec
.%465!     sbc firstFileBlock
```

```
.%466!     bcs +
.%467!     dey
.%468!  +  sta fileClusterCount
.%469!     sty fileClusterCount+1
.%470!     lda clusterBlockCount
.%471!     cmp #2
.%472!     bne +
.%473!     lsr fileClusterCount+1
.%474!     ror fileClusterCount
.%475!
```

Gee, I have more comments embedded in the code than I did last issue.

```
.%476!     ;** load FAT
.%477!  +  lda #<fatbuf
.%478!     ldy #>fatbuf
.%479!     sta 6
.%480!     sty 7
.%481!     lda #1
.%482!     ldy #0
.%483!     ldx fatBlocks
.%484!     jsr copyBlocks
.%485!     bcc +
.%486!     rts
.%487!
.%488!     ;** load actual directory
.%489!  +  lda #<dirbuf
.%490!     ldy #>dirbuf
.%491!     sta 6
.%492!     sty 7
.%493!     lda firstRootDirBlock
.%494!     ldy #0
.%495!     ldx rootDirBlocks
.%496!     jsr copyBlocks
.%497!     bcc +
.%498!     rts
.%499!  +  lda #<dirbuf
.%500!     ldy #>dirbuf
.%501!     ldx rootDirEntries
.%502!     clc
.%503!     rts
.%504!
```

This routine locates the given FAT table entry number and returns the value stored in it. Some work is needed to deal with the 12-bit compressed data structure.

```
.%505!  entryAddr = 2
.%506!  entryWork = 4
.%507!  entryBits = 5
.%508!  entryData0 = 6
.%509!  entryData1 = 7
.%510!  entryData2 = 8
.%511!
.%512!  getFatEntry = *  ;( .AY=fatEntryNumber ) : .AY=fatEntryValue
.%513!     sta entryBits
```

Divide the FAT entry number by two and multiply by three because two FAT entries are stored in three bytes. Then add the FAT base address and we have the address of the three bytes that contain the FAT entry we are interested in. I retrieve the three bytes into zero-page memory for easy manipulation.

```
.%514!     ;** divide by two
.%515!     sty entryAddr+1
.%516!     lsr entryAddr+1
```

```
.%517!     ror
.%518!
.%519!     ;** times three
.%520!     sta entryWork
.%521!     ldx entryAddr+1
.%522!     asl
.%523!     rol entryAddr+1
.%524!     clc
.%525!     adc entryWork
.%526!     sta entryAddr
.%527!     txa
.%528!     adc entryAddr+1
.%529!     sta entryAddr+1
.%530!
.%531!     ;** add base, get data
.%532!     clc
.%533!     lda entryAddr
.%534!     adc #<fatbuf
.%535!     sta entryAddr
.%536!     lda entryAddr+1
.%537!     adc #>fatbuf
.%538!     sta entryAddr+1
.%539!     ldy #2
.%540!  -  lda (entryAddr),y
.%541!     sta entryData0,y
.%542!     dey
.%543!     bpl -
.%544!     lda entryBits
.%545!     and #1
.%546!     bne +
.%547!
```

If the original given FAT entry number is even, then we want the first 12-bit compressed field. The nybbles are extracted according to the diagram shown earlier.

```
.%548!     ;** case 1: first 12-bit cluster
.%549!     lda entryData1
.%550!     and #$0f
.%551!     tay
.%552!     lda entryData0
.%553!     rts
.%554!
```

Otherwise, we want the second 12-bit field.

```
.%555!     ;** case 2: second 12-bit cluster
.%556!  +  lda entryData1
.%557!     ldx #4
.%558!  -  lsr entryData2
.%559!     ror
.%560!     dex
.%561!     bne -
.%562!     ldy entryData2
.%563!     rts
.%564!
```

Finally, this is the file copying level. It deals with reading the clusters of MS-DOS files and copying the data they contain to the already-open CBM Kernal file, possibly with ASCII-to-PETSCII translation.

```
.%565!  ;====file copy level====
.%566!
.%567!  transMode = 14
.%568!  lfn = 15
```

```
.%569!  cbmDataPtr = $60
.%570!  cbmDataLen = $62
.%571!  cluster = $64
.%572!
```

Copy the given cluster to the CBM output file. This routine fetches the next cluster of the file for the next time this routine is called, and if it hits the NULL pointer of the last cluster of a file, it adjusts the number of valid file data bytes the current cluster contains to FileLength % ClusterLength (see note below).

```
.%573!  copyFileCluster = *  ;( cluster, lfn, transMode ) : .CS=err
```

Read the cluster and setup to copy the whole cluster to the CBM file.

```
.%574!     lda cluster
.%575!     ldy cluster+1
.%576!     jsr readCluster
.%577!     bcc +
.%578!     rts
.%579!  +  lda #<clusterBuf
.%580!     ldy #>clusterBuf
.%581!     sta cbmDataPtr
.%582!     sty cbmDataPtr+1
.%583!     lda #0
.%584!     sta cbmDataLen
.%585!     lda clusterBlockCount
.%586!     asl
.%587!     sta cbmDataLen+1
.%588!
```

Fetch the next cluster number of the file, and adjust the cluster data length for the last cluster of the file.

```
.%589!     ;**get next cluster
.%590!     lda cluster
.%591!     ldy cluster+1
.%592!     jsr getFatEntry
.%593!     sta cluster
.%594!     sty cluster+1
.%595!     cmp #$ff
.%596!     bne copyFileClusterData
.%597!     cpy #$0f
.%598!     bne copyFileClusterData
.%599!     lda lenML
.%600!     sta cbmDataLen
.%601!     lda #$01
.%602!     ldx clusterBlockCount
.%603!     cpx #1
.%604!     beq +
.%605!     lda #$03
.%606!  +  and lenML+1
```

The following three lines were added in a last minute panic after realizing that if FileLength % ClusterSize == 0, then the last cluster of the file contains ClusterSize bytes, not zero bytes.

```
.%000!     bne +
.%000!     ldx lenML
.%000!     beq copyFileClusterData
.%607!  +  sta cbmDataLen+1
.%608!
.%609!     copyFileClusterData = *
.%610!     jsr commieOut
.%611!     rts
```

```
.%612!

Copy the file data in the MS-DOS cluster buffer to the CBM output file.

.%613!  cbmDataLimit = $66
.%614!
.%615!  commieOut = *  ;( cbmDataPtr, cbmDataLen ) : .CS=err

If the the logical file number to copy to is 0 ("null device"), then don't
bother copying anything.

.%616!     ldx lfn
.%617!     bne +
.%618!     clc
.%619!     rts

Otherwise, prepare the logical file number for output.

.%620!  +  jsr kernelChkout
.%621!     bcc commieOutMore
.%622!     sta errno
.%623!     rts
.%624!
.%625!     commieOutMore = *

Process the cluster data in chunks of up to 255 bytes or the number of data
bytes remaining in the cluster.

.%626!     lda #255
.%627!     ldx cbmDataLen+1
.%628!     bne +
.%629!     lda cbmDataLen
.%630!  +  sta cbmDataLimit
.%631!     ldy #0
.%632!  -  lda (cbmDataPtr),y
.%633!     bit transMode
.%634!     bpl +

If we have to translate the current ASCII character, look up the PETSCII value
in the translation table and output that value.  If the translation table
entry value is $00, then don't output a character (filter out invalid
character codes).

.%635!     tax
.%636!     lda transBuf,x
.%637!     beq commieNext
.%638!  +  jsr kernelChrout
.%639!     commieNext = *
.%640!     iny
.%641!     cpy cbmDataLimit
.%642!     bne -
.%643!

Increment the cluster buffer pointer and decrement the cluster buffer character
count according to the number of bytes just processed, and repeat the above if
more file data remains in the current cluster.

.%644!     clc
.%645!     lda cbmDataPtr
.%646!     adc cbmDataLimit
.%647!     sta cbmDataPtr
.%648!     bcc +
.%649!     inc cbmDataPtr+1
.%650!  +  sec
.%651!     lda cbmDataLen
```

```
.%652!     sbc cbmDataLimit
.%653!     sta cbmDataLen
.%654!     bcs +
.%655!     dec cbmDataLen+1
.%656!  +  lda cbmDataLen
.%657!     ora cbmDataLen+1
.%658!     bne commieOutMore
```

If we are finished with the cluster, then clear the CBM Kernal output channel.

```
.%659!     jsr kernelClrchn
.%660!     clc
.%661!     rts
.%662!
```

The file copying main routine. Set up for the starting cluster, and call the cluster copying routine until end-of-file is reached. Checks for a NULL cluster pointer in the directory entry to handle zero-length files.

```
.%663!  copyFile = *  ;( startCluster, lenML, .A=transMode, .X=lfn ) : .CS=err
.%664!     ldy #$0e
.%665!     sty $ff00
.%666!     sta transMode
.%667!     stx lfn
.%668!     lda startCluster
.%669!     ldy startCluster+1
.%670!     sta cluster
.%671!     sty cluster+1
.%672!     jmp +
.%673!  -  jsr copyFileCluster
.%674!     bcc +
.%675!     rts
.%676!  +  lda cluster
.%677!     cmp #$ff
.%678!     bne -
.%679!     lda cluster+1
.%680!     cmp #$0f
.%681!     bne -
.%682!     clc
.%683!     rts
.%684!
```

This is the translation table used to convert from ASCII to PETSCII. You can modify it to suit your needs if you wish. If you cannot reassemble this file, then you can sift through the binary file and locate the tabel and change it there. An entry of $00 means the corresponding ASCII character will not be translated. You'll notice that I have set up translations for the following ASCII control characters into PETSCII: Backspace, Tab, Linefeed (CR), and Formfeed. I also translate the non-PETSCII characters such as {, |, ~, and _ according to what they probably would have been if Commodore wasn't so concerned with the graphics characters.

```
.%685!  transBuf = *
.%686!         ;0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f
.%687!  .byte $00,$00,$00,$00,$00,$00,$00,$00,$14,$09,$0d,$00,$93,$00,$00,$00 ;0
.%688!  .byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;1
.%689!  .byte $20,$21,$22,$23,$24,$25,$26,$27,$28,$29,$2a,$2b,$2c,$2d,$2e,$2f ;2
.%690!  .byte $30,$31,$32,$33,$34,$35,$36,$37,$38,$39,$3a,$3b,$3c,$3d,$3e,$3f ;3
.%691!  .byte $40,$c1,$c2,$c3,$c4,$c5,$c6,$c7,$c8,$c9,$ca,$cb,$cc,$cd,$ce,$cf ;4
.%692!  .byte $d0,$d1,$d2,$d3,$d4,$d5,$d6,$d7,$d8,$d9,$da,$5b,$5c,$5d,$5e,$5f ;5
.%693!  .byte $c0,$41,$42,$43,$44,$45,$46,$47,$48,$49,$4a,$4b,$4c,$4d,$4e,$4f ;6
.%694!  .byte $50,$51,$52,$53,$54,$55,$56,$57,$58,$59,$5a,$db,$dc,$dd,$de,$df ;7
.%695!  .byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;8
.%696!  .byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;9
.%697!  .byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;a
```

```
.%698!  .byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;b
.%699!  .byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;c
.%700!  .byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;d
.%701!  .byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;e
.%702!  .byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;f
.%703!
```

This is where the track cache, etc. are stored. This section requires 11K of storage space but does not increase the length of the binary program file since these storage areas are DEFINED rather than allocated with ".buf" directives. The Unix terminology for this type of uninitialized data is "bss".

```
.%704!  ;====bss storage====
.%705!
.%706!  bss = *
.%707!  trackbuf   = bss
.%708!  clusterBuf = trackbuf+4608
.%709!  fatbuf     = clusterBuf+1024
.%710!  dirbuf     = fatbuf+1536
.%711!  end        = dirbuf+4096
```

5. USER-INTERFACE PROGRAM

This section presents the listing of the user-interface BASIC program. You should be aware that you can easily change some of the defaults to your own preferences if you wish. This program is not listed in the ".%nnn!" format that the assembler listing is since you can recover this listing from the uuencoded binary program file. This program should be a little easier to follow than the assembler listing since BASIC is a self-commenting language. :-)

```
10 rem little red reader, by craig bruce, 30-sep-92, for c= hacking netmag
11 :
```

These lines set up the default CBM-DOS and MS-DOS device numbers, taking care to disallow them to be the same device. You can change this to your own drive configuration.

```
20 cd=peek(186)  : rem ** default cbm-dos drive **
25 dv=9:dt=0  :  rem ** ms-dos drive, type (0=1571,255=1581)
26 if dv=cd then dv=8:dt=0 : rem ** alternate ms-dos drive
27 :
30 print chr$(147);"initializing..." : print
40 bank0 : pk=dec("8000")
50 if peek(pk+9)=dec("cb") and peek(pk+10)=131 then 60
55 print"loading machine language routines..." : bload"lrr.bin",u(cd)
60 poke pk+16,dv : poke pk+17,dt : sys pk
```

I "dim" the following variables before the arrays to avoid the overhead of pushing the arrays around when creating new scalar variables.

```
70 dim t,r,b,i,a$,c,dt$,fl$,il$,x,x$
80 dim di$(128),cl(128),sz(128)
90 if dt=255 then dt$="1581" :else dt$="1571"
100 fl$=chr$(19)+chr$(17)+chr$(17)+chr$(17)+chr$(17)
110 il$=fl$:fori=1to19:il$=il$+chr$(17):next
120 goto 500
130 :
131 rem ** load ms-dos directory **
140 print"loading directory..." : print
150 sys pk : sys pk+3
160 dl=0
```

The "rreg" instruction returns the return values of the .A, .X, .Y, and .S registers from the last "sys" call. I check the 1-bit of the .S register (the Carry flag) for error returns.

```
170 rreg bl,dc,bh,s : e=peek(pk+15)
180 if (s and 1) then gosub 380 : return
190 print"scanning directory..." : print
200 db=bl+256*bh
210 if dc=0 then 360
220 for dp=db to db+32*(dc-1) step 32
230 if peek(dp)=0 or peek(dp)=229 then 350
240 if peek(dp+12) and 24 then 350
250 dl=dl+1

This next line is where I set the default selection status, translation type,
and CBM file type for the MS-DOS files.  You can change these defaults simply
by overtyping the string in   (  |  |||  |||  ) the "V" locations.
                                  V  VVV  VVV
260 d$=right$(" "+str$(dl),3)+"     asc  seq  " : rem ** default sel/tr/ft **
270 a$="" : fori=0to10 : a$=a$+chr$(peek(dp+i)) : next
280 a$=left$(a$,8)+"  "+right$(a$,3)
290 print dl; a$
300 d$=d$+a$+"  "
310 cl(dl)=peek(dp+26)+256*peek(dp+27)
320 sz=peek(dp+28)+256*peek(dp+29)+65536*peek(dp+30)
330 di$(dl)=d$+right$("    "+str$(sz),6)
340 sz(dl)=sz
350 next dp
360 return
370 :
371 rem ** report ms-dos disk error **
380 print chr$(18);"ms-dos disk error #";mid$(str$(e),2);
390 print " ($";mid$(hex$(e),3);"), press key.";chr$(146)
400 getkey a$ : return
410 :
411 rem ** screen heading **
420 printchr$(147);"ms-dev=";mid$(str$(dv),2);"    ms-type=";dt$;
430 print"    cbm-dev=";mid$(str$(cd),2):print
440 return
450 :
451 rem ** screen footing **
460 print il$;"d=directory  m=ms-dev  f=cbm-dev q=quit"
470 print"t=toggle-column, c=copy-files, +/- page";
480 return
490 :
491 rem ** main routine **
500 t=1 : c=0
510 r=0
520 gosub 420
530 print "num  s  trn  typ  filename  ext  length"
540 print "---  -  ---  ---  --------  ---  ------"
550 gosub 460
560 b=t+17 : if b>dl then b=dl
570 print fl$;: if t>dl then 590
580 for i=t to b : print di$(i) : next
590 if dl=0 then print chr$(18);"<no files>";chr$(146)
600 if dl=0 then 660
610 print left$(il$,r+5);chr$(18);
620 on c+1 goto 630,640,650
630 print spc(4);mid$(di$(t+r),5,3) : goto 660
640 print spc(7);mid$(di$(t+r),8,5) : goto 660
650 print spc(12);mid$(di$(t+r),13,5) : goto 660
660 getkey a$

Oh shi^Hoot.  I screwed up the following line in the string after the
"+chr$(13)+" part.  You'll notice that I have avoided putting cursor control
characters into the strings everywhere else, but I forgot to do that here.
The "{stuff}" should be CursorUp, CursorDown, CursorLeft, CursorRight,
```

CursorHome, and CursorCLR control characters, respectively. These characters
give the index for the "on" statement below.

```
670 i=instr("dmftc+-q "+chr$(13)+"{stuff}",a$)
680 print left$(il$,r+5);di$(t+r)
690 if i=0 then 600
700 onigoto760,1050,1110,950,1150,1000,1020,730,860,860,770,790,810,830,850,500
710 stop
720 :
721 rem ** various menu options **
730 print chr$(147);"have an awesome day."
740 end
760 gosub 420 : gosub 140 : goto 500
770 r=r-1 : if r<0 then r=b-t
780 goto 600
790 r=r+1 : if t+r>b then r=0
800 goto 600
810 c=c-1 : if c<0 then c=2
820 goto 600
830 c=c+1 : if c>2 then c=0
840 goto 600
850 r=0 : c=0 : goto 600
860 if dl=0 then 600
870 x=t+r : on c+1 gosub 890,910,930
880 print left$(il$,r+5);di$(x) : goto 600
890 if mid$(di$(x),6,1)=" " then x$="*" :else x$=" "
900 mid$(di$(x),6,1)=x$ : return
910 if mid$(di$(x),9,1)="a" then x$="bin" :else x$="asc"
920 mid$(di$(x),9,3)=x$ : return
930 if mid$(di$(x),14,1)="s" then x$="prg" :else x$="seq"
940 mid$(di$(x),14,3)=x$ : return
950 if dl=0 then 600
960 for x=1 to dl
970 on c+1 gosub 890,910,930
980 next x
990 goto 520
1000 if b=dl then t=1 : goto 510
1010 t=t+18 : goto 510
1020 if t=1 then t=dl-(dl-int(dl/18)*18)+1 : goto 510
1030 t=t-18 : if t<1 then t=1
1040 goto 510
1050 print il$;chr$(27);"@";
1060 input"ms-dos device number (8-30)";dv
1061 if cd=dv then print"ms-dos and cbm-dos devices must be different!":goto1060
1070 input"ms-dos device type  (71/81)";x
1080 if x=8 or x=81 or x=1581 then dt=255:dt$="1581" :else dt=0:dt$="1571"
1090 poke pk+16,dv : poke pk+17,dt : sys pk
1100 goto 520
1110 print il$;chr$(27);"@";
1120 input "cbm-dos device number (0-30)";cd
1130 if cd=dv then print"ms-dos and cbm-dos devices must be different!":goto1120
1140 goto 520
1141 :
1142 rem ** copy files **
1150 print chr$(147);"copy files":print:print
1160 if dl=0 then fc=0 : goto 1190
1170 fc=0 : for f=1 to dl : if mid$(di$(f),6,1)="*" then gosub 1200
1180 next f
1190 print : print"files copied =";fc;" - press key"
1191 getkey a$ : goto 520
1200 fc=fc+1
1210 x$=mid$(di$(f),19,8)+"."+mid$(di$(f),29,3)
1220 cf$="":fori=1tolen(x$):if mid$(x$,i,1)<>" " then cf$=cf$+mid$(x$,i,1)
1230 next
1231 if right$(cf$,1)="." then cf$=left$(cf$,len(cf$)-1)
```

```
1232 cf$=cf$+","+mid$(di$(f),14,1)
1240 print str$(fc);". ";chr$(34);cf$;chr$(34);tab(20);sz(f)"bytes";
1245 print tab(35);mid$(di$(f),9,3)
1250 cl=cl(f) : lb=sz(f) - int(sz(f)/65536)*65536
```

I had to use a DOPEN statement here for disk files because the regular OPEN statment does not redirect the DS and DS$ pseudo-variables. You'll notice that the non-disk OPEN statment below has a secondary address of 7. This is to put the printer into lowercase mode if you are outputting directly to it. You can replace this with a 5 (or whatever) if you have a special interface to an IBM-compatible printer and you want to print directly in ASCII. In this case, you would select the "BIN" translation mode for the file you are routing directly to the printer.

```
1260 if cd>=8 then dopen#1,(cf$+",w"),u(cd) :else if cd<>0 then open 1,cd,7
1265 if cd<8 then 1288
1270 if ds<>63 then 1288
1275 x$="y" : print "file exists; overwrite (y/n)";
1280 close 1 : input x$ : if x$="n" then fc=fc-1 : return
1285 scratch(cf$),u(cd)
1286 dopen#1,(cf$+",w"),u(cd)
1288 if cd<8 then 1320
1300 if ds<20 then 1320
1310 print chr$(18)+"cbm disk error: "+ds$ : fc=fc-1 : close1 : return
1320 poke pk+19,cl/256 : poke pk+18,cl-peek(pk+19)*256
1330 poke pk+21,lb/256 : poke pk+20,lb-peek(pk+21)*256
1340 tr=0 : if mid$(di$(f),9,1)="a" then tr=255
1346 x=1 : if cd=0 then x=0
1350 sys pk+6,tr,x
1355 rreg x,x,x,s : e=peek(pk+15)
1356 if (s and 1) then gosub 380 : fc=fc-1
1360 if cd<>0 and cd<8 then close1
1370 if cd>=8 then dclose#1 : if ds>=20 then 1310
1380 return
```

6. UUENCODED FILES

Here are the binary executables in uuencoded form. The CRC32s of the two files are as follows:

```
   "lrr.128"    1106058594
   "lrr.bin"     460671650
```

The "lrr.128" file is the main BASIC program and the "lrr.bin" file contains the machine lanugage disk-accessing routines.

```
begin 640 lrr.128
M`1Q+'`H`CR!,25143$4@4D5$(%)%041%4BP@0ED@0U)!24<@0E)50T4L(#,P
M+5-%4"TY,BP@1D]2($,]($A!0TM)3D<@3D5434%'`%$<"P`Z`(`<%`!#1++"
M*#$X-BD@(#H@CR`J*B!$149!54Q4($-"32U$3U,@1%))5D4@*BH`O!P9`$16
MLCDZ1%2R,"`@.B`@CR`J*B!-4RU$3U,@1%))5D4L(%194$4@*#`],34W,2PR
M-34],34X,2D`\AP:`(L@1%:R0T0@IR!$5K(X.D14LC`@.B"/("HJ($%,5$52
M3D%412!-4RU$3U,@1%))5D4`^!P;`#H`&QT>`)D@QR@Q-#<I.R))3DE424%,
M25I)3D<N+BXB(#H@F0`R'2@`_@(P(#H@4$NRT2@B.#`P,"(I`%P=,@"+(,(H
M4$NJ.2FRT2@B0T(B*2"O(,(H4$NJ,3`ILC$S,2"G(#8P`)P=-P"9(DQ/041)
M3D<@34%#2$E.12!,04Y'54%'12!23U5424Y%4RXN+B(@.B#^$2),4E(N0DE.
M(BQ5*$-$*0"_'3P`ER!02ZHQ-BQ$5B`Z()<@4$NJ,3<L1%0@.B">(%!+`.,=
M1@"&(%0L4BQ"+$DL020L0RQ$5"0L1DPD+$E,)"Q8+%@D``(>4`"&($1))"@Q
M,C@I+$-,*#$R."DL4UHH,3(X*0`J'EH`BR!$5+(R-34@IR!$5"2R(C$U.#$B
M(#K5($14)+(B,34W,2(`4!YD`$9,)++'*#$Y*:K'*#$W*:K'*#$W*:K'*#$W
M*:K'*#$W*0!T'FX`24PDLD9,)#J!2;(QI#$Y.DE,)+))3"2JQR@Q-RDZ@@!^
M'G@`B2`U,#``A!Z"`#H`IAZ#`(\@*BH@3$]!1"!-4RU$3U,@1$E214-43U)9
M("HJ`,8>C`"9(DQ/041)3D<@1$E214-43U)9+BXN(B`Z()D`V!Z6`)X@4$L@
M.B">(%!+JC,`X1Z@`$1,LC```!^J`/X)($),+$1#+$)(+%,@.B!%LL(H4$NJ
M,34I`!H?M`"+("A3(*\@,2D@IR"-(#,X,"`Z((X`.Q^^`)DB4T-!3DY)3D<@
```

```
M1$E214-43U)9+BXN(B`Z()D`3!_(`$1"LD),JC(U-JQ"2`!='](`BR!$0[(P
M(*<@,S8P`'T?W`"!($10LD1"(*0@1$*J,S*L*$1#JS$I(*D@,S(`G1_F`(L@
MPBA$4"FR,""P(,(H1%`ILC(R.2"G(#,U,`"W'_``BR#"*$10JC$R*2"O(#(T
M(*<@,S4P`,,?^@!$3+)$3*HQ``<@!`%$)++)*"(@(JK$*$1,*2PS*:HB("`@
M("!!4T,@(%-%42`@(B`Z((\@*BH@1$5&055,5"!314PO5%(O1E0@*BH`,B`.
M`4$DLB(B(#H@@4FR,*0Q,"`Z($$DLD$DJL<HPBA$4*I)*2D@.B""`$X@&`%!
M)++(*$$D+#@IJB(@("*JR2A!)"PS*0!;("(!F2!$3#L@020`;2`L`40DLD0D
MJD$DJB(@("(`CB`V`4-,*$1,*;+"*$10JC(V*:HR-3:LPBA$4*HR-RD`NB!`
M`5-:LL(H1%"J,C@IJC(U-JS"*$10JC(Y*:HV-34S-JS"*$10JC,P*0#;($H!
M1$DD*$1,*;)$)*K)*"(@("`@(JK$*%-:*2PV*0#I(%0!4UHH1$PILE-:`/(@
M7@&"($10`/@@:`&.`/X@<@$Z`",A<P&/("HJ(%)%4$]25"!-4RU$3U,@1$E3
M2R!%4E)/4B`J*@!0(7P!F2#'*#$X*3LB35,M1$]3($1)4TL@15)23U(@(R([
MRBC$*$4I+#(I.P!](88!F2`B("@D(CO**-(H12DL,RD[(BDL(%!215-3($M%
M62XB.\<H,30V*0"+(9`!H?D@020@.B".`)$AF@$Z`*PAFP&/("HJ(%-#4D5%
M3B!(14%$24Y'("HJ`.$AI`&9QR@Q-#<I.R)-4RU$158](CO**,0H1%8I+#(I
M.R(@("`@35,M5%E013TB.T14)#L``B*N`9DB("`@($-"32U$158](CO**,0H
M0T0I+#(I.ID`""*X`8X`#B+"`3H`*2+#`8\@*BH@4T-2145.($9/3U1)3D<@
M*BH`72+,`9D@24PD.R)$/41)4D5#5$]262`@33U-4RU$158@($8]0T)-+41%
M5B!1/5%5250B`(TBU@&9(E0]5$]'1TQ%+4-/3%5-3BP@0SU#3U!9+49)3$53
M+"`K+RT@4$%'12([`),BX`&.`)DBZ@$Z`+(BZP&/("HJ($U!24X@4D]55$E.
M12`J*@#`(O0!5+(Q(#H@0[(P`,@B_@%2LC``TB((`HT@-#(P``(C$@*9(").
M54T@(%,@(%123B`@5%E0("!&24Q%3D%-12`@15A4("!,14Y'5$@B`#(C'`*9
M("(M+2T@("T@("TM+2`@+2TM("`M+2TM+2TM+2`@+2TM("`M+2TM+2TB`#PC
M)@*-(#0V,`!7(S`"0K)4JC$W(#H@BR!"L41,(*<@0K)$3`!P(SH"F2!&3"0[
M.B"+(%2Q1$P@IR`U.3``C2-$`H$@2;)4(*0@0B`Z()D@1$DD*$DI(#H@@@"V
M(TX"BR!$3+(P(*<@F2#'*#$X*3LB/$Y/($9)3$53/B([QR@Q-#8I`,<C6`*+
M($1,LC`@IR`V-C``WR-B`ID@R"A)3"0L4JHU*3O'*#$X*3L`]R-L`I$@0ZHQ
M((D@-C,P+#8T,"PV-3``&21V`ID@IC0I.\HH1$DD*%2J4BDL-2PS*2`Z((D@
M-C8P`#LD@`*9(*8W*3O**$1))"A4JE(I+#@L-2D@.B")(#8V,`!?)(H"F2"F
M,3(I.\HH1$DD*%2J4BDL,3,L-2D@.B")(#8V,`!I))0"H?D@020`D"2>`DFR
MU"@B1$U&5$,K+5$@(JK'*#$S*:HBD1&='1.3(BQ!)"D`JB2H`ID@R"A)3"0L
M4JHU*3M$220H5*I2*0"Z)+("BR!)LC`@IR`V,#``!B6\`I%)B3<V,"PQ,#4P
M+#$Q,3`L.34P+#$Q-3`L,3`P,"PQ,#(P+#<S,"PX-C`L.#8P+#<W,"PW.3`L
M.#$P+#@S,"PX-3`L-3`P``PEQ@*0`!(ET`(Z`#,ET0*/("HJ(%9!4DE/55,@
M345.52!/4%1)3TY3("HJ`%<EV@*9(,<H,30W*3LB2$%612!!3B!!5T533TU%
M($1!62XB`%TEY`*``'<E^`*-(#0R,"`Z((T@,30P(#H@B2`U,#``D24"`U*R
M4JLQ(#H@BR!2LS`@IR!2LD*K5`";)0P#B2`V,#``M246`U*R4JHQ(#H@BR!4
MJE*Q0B"G(%*R,`"_)2`#B2`V,#``UR4J`T.R0ZLQ(#H@BR!#LS`@IR!#LC(`
MX24T`XD@-C`P`/DE/@-#LD.J,2`Z((L@0[$R(*<@0[(P``,F2`.)(#8P,``9
M)E(#4K(P(#H@0[(P(#H@B2`V,#``*B9<`XL@1$RR,""G(#8P,`!*)F8#6+)4
MJE(@.B"1($.J,2"-(#@Y,"PY,3`L.3,P`&HF<`.9(,@H24PD+%*J-2D[1$DD
M*%@I(#H@B2`V,#``E29Z`XL@RBA$220H6"DL-BPQ*;(B("(@IR!8)+(B*B(@
M.M4@6"2R(B`B`*XFA`/**$1))"A8*2PV+#$ILE@D(#H@C@#=)HX#BR#**$1)
M)"A8*2PY+#$ILB)!(B"G(%@DLB)"24XB(#K5(%@DLB)!4T,B`/8FF`/**$1)
M)"A8*2PY+#,ILE@D(#H@C@`F)Z(#BR#**$1))"A8*2PQ-"PQ*;(B4R(@IR!8
M)+(B4%)'(B`ZU2!8)+(B4T51(@!`)ZP#RBA$220H6"DL,30L,RFR6"0@.B".
M`%$GM@.+($1,LC`@IR`V,#``8"?``X$@6+(Q(*0@1$P`>"?*`Y$@0ZHQ((T@
M.#DP+#DQ,"PY,S``@"?4`X(@6`"*)]X#B2`U,C``HR?H`XL@0K)$3""G(%2R
M,2`Z((D@-3$P`+8G\@-4LE2J,3@@.B")(#4Q,`#B)_P#BR!4LC$@IR!4LD1,
MJRA$3*NU*$1,K3$X*:PQ."FJ,2`Z((D@-3$P`/LG!@14LE2K,3@@.B"+(%2S
M,2"G(%2R,0`%*!`$B2`U,3``&B@:!)D@24PD.\<H,C<I.R)`(CL`0"@D!(4B
M35,M1$]3($1%5DE#12!.54U"15(@*#@M,S`I(CM$5@"%*"4$BR!#1+)$5B"G
M()DB35,M1$]3($%.1"!#0DTM1$]3($1%5DE#15,@35535"!"12!$249&15)%
M3E0A(CJ),3`V,`"J*"X$A2)-4RU$3U,@1$5624-%(%194$4@("@W,2\X,2DB
M.U@`ZR@X!(L@6+(X(+`@6+(X,2"P(%BR,34X,2"G($14LC(U-3I$5"2R(C$U
M.#$B(#K5($14LC`Z1%0DLB(Q-3<Q(@`.*4($ER!02ZHQ-BQ$5B`Z()<@4$NJ
M,3<L1%0@.B">(%!+`!@I3`2)(#4R,``M*58$F2!)3"0[QR@R-RD[(D`B.P!5
M*6`$A2`B0T)-+41/4R!$159)0T4@3E5-0D52("@P+3,P*2([0T0`FBEJ!(L@
M0T2R1%8@IR"9(DU3+41/4R!!3D0@0T)-+41/4R!$159)0T53($U54U0@0D4@
M1$E&1D5214Y4(2(ZB3$Q,C``I"ET!(D@-3(P`*HI=00Z`,$I=@2/("HJ($-/
M4%D@1DE,15,@*BH`WRE^!)D@QR@Q-#<I.R)#3U!9($9)3$53(CJ9.ID`^BF(
M!(L@1$RR,""G($9#LC`@.B")(#$Q.3``+RJ2!$9#LC`@.B"!($:R,2"D($1,
M(#H@BR#**$1))"A&*2PV+#$ILB(J(B"G((T@,3(P,``W*IP$@B!&`&,JI@29
M(#H@F2)&24Q%4R!#3U!)140@/2([1D,[(B`M(%!215-3($M%62(`=2JG!*'Y
M($$D(#H@B2`U,C``@2JP!$9#LD9#JC$`JBJZ!%@DLLHH1$DD*$8I+#$Y+#@I
```

```
MJB(N(JK**$1))"A&*2PR.2PS*0#E*L0$0T8DLB(B.H%)LC&DPRA8)"DZBR#*
M*%@D+$DL,2FSL2(@(B"G($-&)+)#1B2JRBA8)"Q)+#$I`.LJS@2"`!0KSP2+
M(,DH0T8D+#$ILB(N(B"G($-&)++(*$-&)"S#*$-&)"FK,2D`,RO0!$-&)+)#
M1B2J(BPBJLHH1$DD*$8I+#$T+#$I`&<KV`29(,0H1D,I.R(N("([QR@S-"D[
M0T8D.\<H,S0I.Z,R,"D[4UHH1BDB0EE415,B.P"`*]T$F2"C,S4I.\HH1$DD
M*$8I+#DL,RD`KROB!$-,LD-,*$8I(#H@3$*R4UHH1BD@JR"U*%-:*$8IK38U
M-3,V*:PV-34S-@#I*^P$BR!#1+&R.""G(/X-(S$L*$-&)*HB+%<B*2Q5*$-$
M*2`ZU2"+($-$L[$P(*<@GR`Q+$-$+#<`^ROQ!(L@0T2S.""G(#$R.#@`#RSV
M!(L@1%.SL38S(*<@,3(X.``^+/L$6"2R(EDB(#H@F2`B1DE,12!%6$E35%,[
M($]615)74DE412`H62].*2([`&8L``6@(#$@.B"%(%@D(#H@BR!8)+(B3B(@
MIR!&0[)&0ZLQ(#H@C@!W+`4%\BA#1B0I+%4H0T0I`)$L!@7^#2,Q+"A#1B2J
M(BQ7(BDL52A#1"D`HRP(!8L@0T2S.""G(#$S,C``MBP4!8L@1%.S,C`@IR`Q
M,S(P`.PL'@69(,<H,3@IJB)#0DT@1$E32R!%4E)/4CH@(JI$4R0@.B!&0[)&
M0ZLQ(#H@H#$@.B".`!DM*`67(%!+JC$Y+$-,K3(U-B`Z()<@4$NJ,3@L0TRK
MPBA02ZHQ.2FL,C4V`$8M,@67(%!+JC(Q+$Q"K3(U-B`Z()<@4$NJ,C`L3$*K
MPBA02ZHR,2FL,C4V`&XM/`544K(P(#H@BR#**$1))"A&*2PY+#$ILB)!(B"G
M(%12LC(U-0"%+4(%6+(Q(#H@BR!#1+(P(*<@6+(P`)4M1@6>(%!+JC8L5%(L
M6`"Q+4L%_@D@6"Q8+%@L4R`Z($6RPBA02ZHQ-2D`T2U,!8L@*%,@KR`Q*2"G
M((T@,S@P(#H@1D.R1D.K,0#I+5`%BR!#1+.Q,""O($-$LS@@IR"@,0`.+EH%
IBR!#1+&R.""G(/X/(S$@.B"+($13L;(R,""G(#$S,3``%"YD!8X`````
`
end

begin 640 lrr.bin
M`(!,68%,3H),#H3+@P`````````````````````````````````````2*D`
MA="M$(`@L?^I;R"3_ZE5(*C_)-`P#ZDP(*C_:""H_R30,`(88*D%C0^`.&"M
M`-U)$(T`W6"I""P-W/#[8"!@@*X,W"!7@(I@J1H@*("0`6`@KO\DT##.&"!'
M_RP-W"!7@"!H@(T/@"D/R0*P):``(&B`F2*`R,`&D/48J0@@*("0`6"I`2P1
M@#`"J00@J/\@KO]@2(HI`0H*"@HL$8`0`DD0("B`D`%@:""H_ZD!(*C_J0D@
MJ/\@KO]X&"!'_RP-W"!7@*DXH(6%`H0#J0"%!"P1@#`#(!.!("N!L`GF!*4$
MR0F0ZQA88*DXA0*F!!BIA7TB@84#8``($`8.!`P""B!H@(T/@"D/R0*0`6"B
M`J``J0@L#=SP^ZT`W4D0C0#=K0S<D0+(T.GF`\K0Y&"I#HT`_ZG_C2"`C2&`
M&&#-((#0$.PA@-`+B)@*&&F%J*DX&&"-(((".(8"$!2"^@)`9K0^`*0_)"_`"
M.&`@<X"M(("N(8"D!9#;8*T@@*XA@*0%3&B!HO_H..D2L/J($/<8:1+(J(I@
M(*Z!H@#`"9`(2)CI":AHH@'(8(8+(,"!A0B&"80*I0BF":0*(&B!D`%@A0R$
M#:("H`"Q#)$&R+$,D0;(T/3F#>8'RM#MY@JE"LD*D!*I`84*Y@FE"<D"D`:I
M`(4)Y@C&"]"[&&`XZ0*P`8BN%H#@`?`'"H0')@>D!QAM'("0`<BB.(8&HI>&
M!ZX6@$S3@:D.C0#_J0"@`"#`@2!H@9`!8(4"A`.@#;$"C1:`R0.0!ZD\C0^`
M.&"@$+$"R0+0\:`6L0*-%X#)!+#FH!&Q`HT9@,F!L-M*2DI*C1B`H!.Q`HT:
M@,BQ`HT;@*`8L0+)"="_H!JQ`LD"T+>@#K$"R0'0KZT7@`H8:0&-'8`8;1B`
MC1R`K1J`K!N`..T<@+`!B(T>@(P?@*T6@,D"T`9.'X!N'H"I.*";A0:$!ZD!
MH`"N%X`@TX&0`6"I.*"AA0:$!ZT=@*``KAB`(-.!D`%@J3B@H:X9@!A@A06$
M`T8#:H4$I@,*)@,8902%`HIE`X4#&*4":3B%`J4#:9N%`Z`"L0*9!@"($/BE
M!2D!T`BE!RD/J*4&8*4'H@1&"&K*T/JD"&"E9*1E("6"D`%@J3B@EX5@A&&I
M`(5BK1:`"H5CI62D92`E@X5DA&7)_]`@P`_0'*T4@(5BJ0&N%H#@`?`"J0,M
M%8#0!:X4@/`"A6,@OH-@I@_0`AA@(,G_D`2-#X!@J?^F8]`"I6*%9J``L6`D
M#A`&JKTXA/`#(-+_R,1FT.P8I6!E9H5@D`+F83BE8N5FA6*P`L9CI6(%8]#$
M(,S_&&"@#HP`_X4.A@^M$H"L$X"%9(1E3"J$(&^#D`%@I63)_]#TI67)#]#N
M&&`````````!0)#0"3`````````````````````````"`A(B,D)28G*"DJ
M*RPM+B\P,3(S-#4V-S@Y.CL\/3X_0,'"P\3%QL?(R<K+S,W.S]#1TM/4U=;7
MV-G:6UQ=7E_`04)#1$5&1TA)2DM,34Y/4%%24U155E=865K;W-W>WP`````
M`````````````````````````````````````````````````````````````
M`````````````````````````````````````````````````````````````
A`````````````````````````````````````````````
`
end
```

7. BIBLIOGRAPHY

The following works were consulted in creating this article:

[1] Jim Butterfield, "Jim Butterfield's Complete C128 Memory Map",
    _The_Transactor_, Volume 7, Issue 01, July 1986 (A Must!).

[2] Commodore Business Machines, _Commodore_1571_Disk_Drive_User's_Guide_,

CBM, 1985.

[3] Some program called "msdos-to-128" included with "cs-dos" by
M. G-something. Originally published in COMPUTE!'s Gazzette, I think.

[4] Commodore Business Machines, _Commodore_128_Programmer's_Reference_Guide_,
Bantam Books, 1986.

[5] _The_Transactor_, Volume 4, Issue 05 ("The Reference Issue"), May 1983.

==========

Next Issue:

Learning Machine Language - Part 5

The SPACE INVASION is continued with the design and implementation of the player and alien animation along with a look at device scanning for the 1351 mouse, joystick and keyboard.

The 1351 Mouse Demystified

Finally! - After 2 delays, this article will explain how the 1351 mouse works as well as provide a easy to use interface in machine language for both basic and machine language programmers. An example program will be given to illustrate both the 1351 mouse and the multi-tasking system.

Multi-tasking on the C=128

A rudimentary multi-tasking system will be implemented for tasks to run con-currently with each other. While intended for machine language programmers some discussion of how to use this within basic will be given so that more than one basic / ml program can be run at a time. An example program will be given to illustrate both the 1351 mouse and the multi-tasking system.

Stretching sprites

You might have heard that it is possible to expand sprites to more than twice their original size. But there is no need to expand all of them equally. This article will examine on how to expand them 2,3 or more multiples of their original size.

LITTLE RED WRITER: MS-DOS file writer for the 128 and 1571/81 drives.

This article will extend the Little Red Reader program to be able to write Commodore-DOS files to an MS-DOS disk.

==========